Composing Concurrent Objects

Applying Composition Filters
for the Development and Reuse
of Concurrent Object-Oriented Programs

Lodewijk M.J. Bergmans

ISBN 90-9007359-0

illustrations from the *Encyclopédie* by Diderot and d'Alembert (1762-1777)
printed by CopyPrint 2000, Enschede

COMPOSING CONCURRENT OBJECTS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. Th.J.A. Popma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 24 juni 1994 te 15.00 uur.

door
Louis Marie Johannes Bergmans
geboren op 11 mei 1967
te Tilburg

Dit proefschrift is goedgekeurd door:
Prof. dr. ir. A. Nijholt, promotor, en
Dr. ir. M. Aksit, assistent-promotor

Abstract

Adopting the object-oriented paradigm for the development of large and complex software systems offers several advantages, of which increased extensibility and reusability are the most prominent ones. The object-oriented model is also quite suitable for modelling concurrent systems. However, it appears that extensibility and reusability of concurrent applications is far from trivial. In addition, very little attention has been paid by the conventional object-oriented development methodologies to the analysis and design of synchronisation constraints for concurrent objects.

To address these problems, in this thesis the framework of *composition-filters*, an extension to the object-oriented model, is adopted. An analysis is presented of the problems involved in reusing and extending concurrent objects, in particular the so-called *inheritance anomalies*. Based on this analysis, a set of criteria for effective extensible concurrent object-oriented programming languages is formulated.

The thesis introduces techniques for the creation of concurrency and the synchronisation of concurrent activities, fully integrated within the (object-oriented) composition-filters model. Important properties of the proposed object model are: all objects are -potentially- active, intra-object concurrency is supported and synchronisation specifications are fully separated from method implementations. The applicability and expressive power of the proposed technique are demonstrated, and it is shown how reusability and extensibility of concurrent objects are achieved.

The implementation aspects that are involved in the proposed synchronisation scheme are addressed, and some optimisation techniques are presented. It is also shown how the synchronisation specifications can be translated into boolean synchronisation constraints on messages; this is applicable both for implementation purposes and for reasoning about synchronisation specifications.

To employ these techniques for the development of concurrent applications, a software development method is introduced that focuses on the analysis and specification of synchronisation constraints for objects. This includes a graphical notation for specifying synchronisation and a number of method steps and hints for the derivation of reusable synchronisation specifications. The graphical notation can be translated algorithmically to composition-filters synchronisation specifications.

The presented material can aid software engineers in the development of concurrent object-oriented applications in several ways: potential problems in reusing and extending concurrent objects are made explicit and analysed, a technique is presented to overcome these problems, and the analysis and design of synchronisation constraints, with the aid of an intuitive and precise notation, is addressed.

Acknowledgements

The past four years, during which I have been conducting my PhD research, conclude almost 21 years of education. Somehow, I feel that they have been the most fruitful and intensive years of my life. Mehmet Aksit, who has been my supervisor during these years, is largely responsible for this. His experience, enthousiasm for research and incredible energy have provided a continuous stimulus for my research. In addition, he provided fruitful and productive working conditions, and numerous contacts both in the international research community and in industry. Meanwhile he always kept a critical eye on the quality of the research. Mehmet, you have not only been an inspiring supervisor, but also a good friend.

I would also like to thank my promotor, Anton Nijholt for his supervision. He kept an eye on my progress, and his door open whenever I needed his advise. The other members of my PhD comittee; Pierre America, Jan van den Bos, Satoshi Matsuoka, Eddy Michiels, Sape Mullender and Oscar Nierstrasz, all took great effort in reading an earlier draft of this thesis and gave valuable comments.

The members of the TRESE project have always offered me a pleasant working environment. Of the recent project members I would like to mention a few in particular. Jan Bosch, who -unfortunately- recently moved to Sweden. I now miss both an enthousiastic colleague and a good friend. Richard van de Stadt, who has demonstrated remarkable patience in answering and solving hundreds of stupid questions from me; thanks. And Bedir Tekinerdogan, who just recently joined us, and had to cope with someone who never had time, except for bugging him about silly things.

Throughout the years, several other people have been working in the TRESE project: Jan Willem Dijkstra, Chuck Grossman, Ben Hekster, Margriet Offereins, Sander Pool, Jan de Visser, Sinan Vural, Ken Wakita and Eddy Zondag. I enjoyed working together with these people, and learned a lot from each of them. I also took pleasure from the co-operation with the numerous students that did their graduation projects and practical trainings in our project. There are so many of them that I could not even start to mention them one by one, but each of them has left unique traces in my memory.

I owe my gratitude to all the people I met during conferences, workshops and visits the past four years; they have provided me repeatedly with useful comments, inspiring discussions and pleasant company. I would particularly like to thank Akinori Yonezawa, Satoshi Matsuoka and the people in their group, for their hospitality and fruitful discussions during my stay at the University of Tokyo. This has been both a pleasant and enlightening experience for me.

I would not have had the energy for this research if it were not for all the people that offered their warm friendship to me, even though I often neglected them terribly. I would like to thank them all for coping with me. This includes my colleagues from the 'kookgroep'; Gert, Hetty, Jan, Klaas, Marc, Margriet, Marjo, Paul and Wil, thanks for good meals and good company.

Finally, here is a chance to express my gratitude and love for my parents; Marijke and Klaas; you are more important to me than you may think. As for my other relatives; thanks for being my friends.

I conclude with the most important person for me. Ingrid, thanks for your love and support; you have made this work possible. Thanks for making my life complete.

Table of Contents

Chapter 1: Introduction and Background

Chapter 2: The Composition-Filters Object Model

Chapter 3: Concurrency and Synchronisation

Chapter 4: The Analysis and Design of Concurrent Objects

Chapter 5: Implementation Aspects

Chapter 6: Conclusion

Appendices

Preface

A PhD thesis is almost by definition are very specialistic document; most of the research investigates the far corners of the known scientific universe. Indeed the research that is presented here focusses on the construction of *concurrent*, *object-oriented* programs within the *composition-filter* paradigm. This topic is addressed however, for the various phases in software development; from analysis and design up to the execution phase.

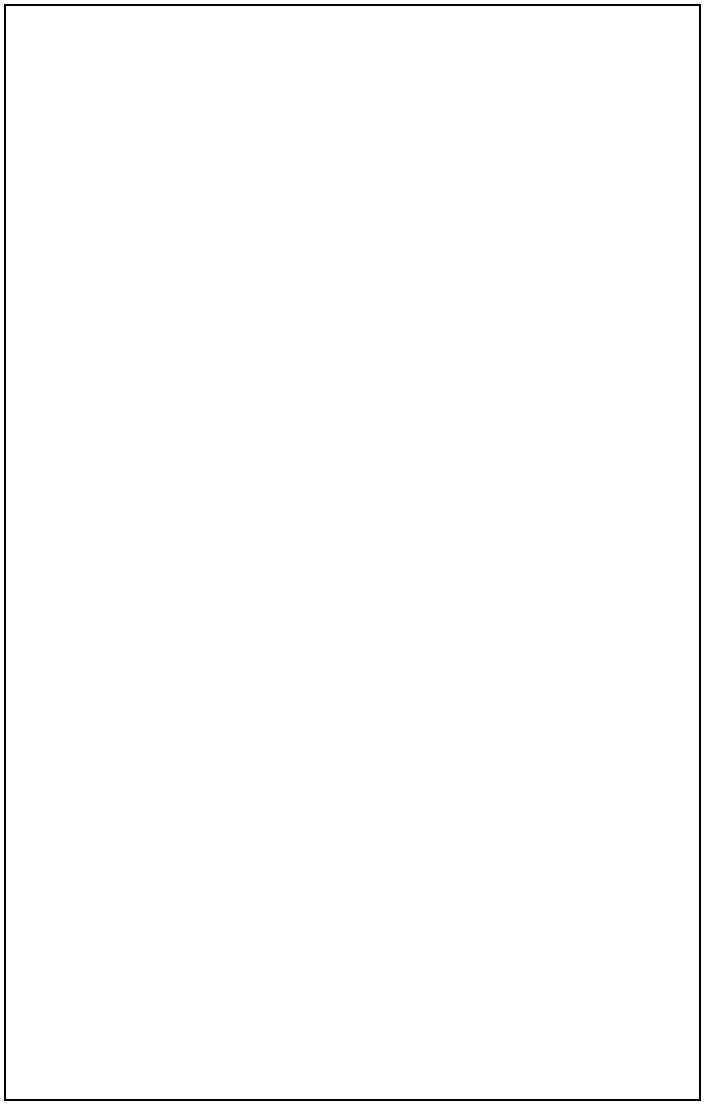
The primary aim of this work is to investigate how concurrent activities and the synchronisation between these can be succesfully modelled in the object-oriented paradigm. The composition-filters model of computation is adopted as a framework for addressing these issues. One of the prime concerns is to ensure the reusability and extensibility of software components; well-defined components should be reusable in various contexts. Through extension components can be tailored for changing requirements or different contexts.

At the core of these modelling issues is a suitable computation model. This is achieved by integrating constructs for concurrency and synchronisation within the composition filters model. The resulting computation model is adopted by the programming language *Sina*.

The modelling and programming language level is only one phase, albeit an essential one, in the development of software. Proper modelling of a system is only feasible once it is known precisely *what* to model and *how* to model it; these are the respective goals of the analysis and the design phase. In this thesis the analysis and design of objects is addressed, focussing on the aspect of synchronisation of activities.

The eventual goal of software development is to come up with an executable specification; the introduction of a programming language cannot completely ignore the implementation aspects involved. Therefore the thesis addresses the issues involved in the realisation of the synchronisation mechanism, and proposes optimisation techniques for improving efficiency.

Chapter 1



Introduction and Background

1.1 The Problem Statement

The problems of concurrent programming and synchronising access to shared resources by concurrent processes have been studied extensively during the past three decades. Initially, these studies were intended to aid the design of multi-programmable operating systems [Dijkstra 68] running on a single processor. With the introduction of parallel computers, high-level language constructs [Kallstrom 88] were required to express parallelism and synchronisation. During the last 15 years, further advancements in software, hardware and communication technology facilitated the use of distributed computers co-operating via local area networks [Coulouris 88]. This motivated language designers to incorporate language structures that are more suitable for distributed programming.

The continuous growth in size and complexity of today's computer systems increase the necessity for new software development methods and language mechanisms that support a high degree of modularity, extensibility and reusability [Meyer 88]. The object-oriented paradigm, supported by software development methods (e.g. OMT [Rumbaugh 91], Booch [Booch 90], OOA [Coad 91a, 91b]) and programming languages (e.g. Smalltalk [Goldberg 83], C++ [Stroustrup 86] and Eiffel [Meyer 88, 92]), lends itself well to these objectives.

In addition, has been claimed that object-oriented language constructs are also suitable for expressing concurrency and synchronisation [Yonezawa 87]. In fact, it has been suggested that the notion of concurrent objects is essential for the construction of distributed and open systems [Tokoro 94]. However, most concurrent object-oriented languages fail in combining their concurrency and synchronisation mechanisms with inheritance. This problem is referred to as the *inheritance anomaly*. Ideally, development methods and language constructs should be able to express concurrency and synchronisation while maintaining a high degree of modularity, extensibility and reusability.

Problem Statement

The aim of this thesis is to investigate the integration of concurrency and synchronisation in object-oriented software development. One of the prime concerns is the provision of suitable software engineering properties. Of these, we focus mainly on the extensibility and reusability of concurrent software. In particular, concurrency should be cleanly integrated with objects without causing interference with other object properties.

Apart from these modelling issues, we consider it important to address the analysis and design of concurrent objects as well. In particular, we are interested in the derivation and specification of synchronisation constraints on objects as a part of an object-oriented software methodology.

The Approach

First of all, the *composition-filters* model for computation is adopted. The composition-filters model provides techniques to manage the complexity in object-oriented systems. It is a modular extension to the conventional object model, which offers composable techniques for specifying the behaviour of the object. It can be viewed as a framework that allows for specifying different aspects of the object behaviour in a rigid, consistent manner. These

aspects include data abstraction techniques such as inheritance and delegation, but also include multiple views or roles, object-oriented queries and real-time constraints.

The composition-filters framework is open-ended in the sense that new aspects of the object behaviour can be added by introducing new primitive *filter types*. Important properties are that the various aspects are *orthogonal*, i.e. they can be freely combined with each other, and that they are integrated with the object-oriented model.

After an analysis of the potential reusability problems for concurrent objects, and in particular of the interference between synchronisation and inheritance, we address the integration of concurrency and synchronisation with the (object-oriented) composition-filters model. A new primitive filter type will be introduced, that allows for specifying synchronisation constraints on objects.

An additional concern is to provide support for the software analysis and design process; an object-oriented software development method is described that focuses on the synchronisation of messages. It is attempted to support the specification of synchronisation constraints in an intuitive manner by describing them with a graphical notation. These synchronisation constraints can be mapped from the graphical notation to the composition-filters model in a straightforward manner. The development of reusable concurrent objects is supported through a number of hints and guidelines.

1.2 Background

As a general background to this thesis the concepts of the object-oriented model and object-oriented software development are introduced. A notation is described for graphically specifying the most important object-oriented concepts.

1.2.1 The Object-Oriented Model

This subsection describes the most important concepts of the object-oriented model.

Objects

An object is a module that combines data and operations working on the data (cf. abstract data types). The data is hidden, or *encapsulated* within the object boundaries. The operations are called *methods*[1] and they constitute the interface of the object. The behaviour of an object is determined by the methods that can be invoked from outside through *message invocations*. A message is a request for an object to perform a particular method. The data, or state, of the object can only be changed through message invocations. The state is constituted by a number of *instance variables*. This is depicted in the following figure:

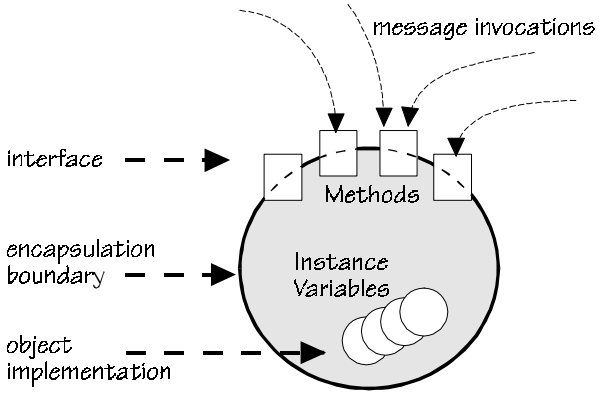


Figure 1.2.1 The Object Model

Encapsulation ensures that the implementation of an object is never visible from the outside; this includes the implementation of methods and the number and types of instance variables. In a pure object language, every variable is an object (cf. Smalltalk-80 [Goldberg 83]). In some languages, such as C++ [Stroustrup 86] or Eiffel [Meyer 88], variables can be both simple data types such as integers and booleans, or fully-fledged objects.

Classes

A *class* is an abstraction of a group of objects which defines their common behaviour. A class also functions as a *template* for creating new objects. These objects are termed

---

[1] The term *method* can refer both to the operations of an object, and to a software development method. The meaning will usually be clear from the context, and in some cases we will use the term operation to avoid confusion.

instances of the class. Each instance of a class has its own copy of the instance variables, whereas the methods of all instances have the same definition.

Inheritance

The mechanism of *inheritance* allows for reusing the specification of a class in the definition of a new class. A class reuses the behaviour of its *superclass* by offering the methods of the superclass on its interface. The *subclass* may add new methods, or redefine (*override*) methods defined by the superclass. Inheritance is a structural relation between classes that provides a second level of abstraction on objects. Inheritance increases manageability by additional structure and improves modelling power by offering a classification relation between components.

Apart from *single inheritance*, where each class has a single superclass, *multiple inheritance* is provided by some systems as well. In the case of multiple inheritance, a class can have multiple superclasses. This can even improve the modelling power and reuse potential. Single inheritance results in a tree-shaped inheritance hierarchy with a single root, whereas multiple inheritance creates a lattice with possibly several roots.

Delegation

Inheritance is a *reuse relation* between classes. *Delegation* is a similar reuse relation between objects; a *delegating* object can re-direct the requests it receives to a *delegated* object. An important property of delegation is that the delegated object must be part of the extended identity of the delegating object. As such, self-reference within the delegated object must designate the delegating object. Ensuring this property, which further distinguishes delegation from message sending, is referred to as the *self-problem* [Lieberman 86].

It is claimed that delegation is more powerful than inheritance, because (a) delegation can simulate inheritance [Sciore 86], and (b) delegation supports the dynamic evolution of systems, whereas inheritance relations are statically defined relations between classes. A further distinction is that a delegating object can share both the behaviour *and* the state of the delegated object, whereas a subclass only inherits behaviour from its superclass(es).

Polymorphism

The property of polymorphism distinguishes message sending from a function call. Through polymorphism different objects can respond differently to the same message. This is an important property, as it allows for defining abstract protocols for interaction between objects, regardless of specific object identity. Polymorphism through inheritance is the redefinition of method implementations in subclasses.

Self-Reference

The polymorphic pseudo-variable *self* (also termed *this*, *current*, or *server* in various programming languages) provides a means for self-reference. Self provides a reference to the object that received the request, even when that request is performed by a method that has been inherited from another class or delegated to another object. As a result, sending a message to *self* may cause the execution of a method that is (re-)defined by a subclass. This

provides a form of open-endedness that can be used for tailoring and extending the behaviour of an object without modifying its definition.

Classification of Object Models

In [Wegner 87, 90] the following classification of languages that adopt object models is presented: an *object-based* language provides objects, i.e. abstract data types that encapsulate the implementation and are accessed through polymorphic message invocations. Adding the concept of classes to objects results in a *class-based* language. A truly *object-oriented* language offers both objects, classes and inheritance. Obviously, an object-oriented language is also a class-based language, and a class-based language is automatically object-based.

Examples of object-based languages are Ada [Ada 80], SELF [Ungar 87] and Actor-languages [Agha 88]. An example of a class-based language is CLU [Liskov 77]. Famous examples of object-oriented languages are Simula [Birtwistle 73], Smalltalk-80 [Goldberg 83], C++ [Stroustrup 86] and Eiffel [Meyer 88].

Types of Relations between Objects

The relationships between objects can be classified into three categories: *reuse* relations, *aggregation* relations and *usage* relations. A reuse relation[2] can represent both (multiple) inheritance and delegation. An aggregation relation is the containment -or encapsulation- of objects within another object, as realised by instance variables. A usage relation from one object to another indicates that the first object sends messages to the second.

1.2.2 Object-Oriented Software Development

The target of object-oriented software development is the *object-oriented decomposition* of user's needs into executable language constructs. The object-oriented decomposition process can be sub-divided into *analysis*, *design* and *implementation* phases.

In the analysis phase, the software engineer aims at precise and correct identification and specification of the user's needs in an *understandable* way. This phase is mainly directed by the user's problem, or the so called *real-world domain*. In the design phase, the software engineer revises and extends the analysis model by specifying how the user requirements can be realised. The implementation phase details the design model by means of specific language constructs.

An important characteristic of object-oriented development is that the analysis, design and implementation phases adopt similar models, although each phase has a different emphasis. This enables a smooth transition between the different phases. Each phase in object-oriented software development can be divided into three sub-components: *preparatory work*, *structural relations* and *object interactions*.

---

[2] We distinguish a *reuse* relationship from a *use* relationship. The latter, which is implemented by message connections, can be viewed as the passive form of reuse: there is no added value to the server object, whereas in a reuse relationship, the client object tries to build an extension to the server object. The latter may even require redefinition of some of the behaviour of the server object.

Preparatory Work

The preparatory work in the analysis phase consists of mapping between the *real world* entities and the entities in the analysis model: objects[3]. This mapping process is called *domain analysis*. Another important activity is the partitioning of the problem domain into manageable sub-components called *subsystems*.

As an example of the application of real-world knowledge, most theory books introduce classification hierarchies to organise knowledge. These hierarchies usually can be directly represented as object-oriented class hierarchies. Since the basic aim of theory is to introduce sound and generic solutions, the software engineer can then create highly reusable inheritance hierarchies.

The preparatory work for the design phase consists mainly of mapping the analysis model to a design environment. In the design environment, libraries of predefined classes may be available. The objects identified in the analysis phase are mapped, as far as possible, to these predefined classes. Another preparatory activity of the design phase is collecting and formulating design requirements which may not have been relevant in the analysis phase. For example, efficiency and alternative realisations are typical design requirements.

The preparatory work for the implementation phase is concerned with the environment within which the design will be implemented. For example, the properties and restrictions of the implementation language may require a non-trivial mapping between the design model and the language model.

Structural Relations

Object-oriented analysis concentrates on a few specific types of relations. The two most important relations are the *classification* and *part-of* relations. These correspond respectively to the reuse relations and the aggregation relations of the object-oriented model. In addition, some methods ([Coad 91a] and [Rumbaugh 91]) introduce *associations* which describe relations other than classification and part-of relations among classes.

Classification relations indicate that one class may be considered as a generalisation or specialisation of another class. This is a common way of making abstractions, resulting in classification trees, with the most general cases at the root of the tree, and the most specialised and dedicated cases as the leaves.

Part-of relations may reflect part-whole relations such as wheels are part of a whole car, or for example, organisations consist of departments. Subsystem partitioning in the analysis phase often matches part-of relations.

In the design and implementation phases, classification structures manifest themselves as class-inheritance and delegation hierarchies. During design and implementation, the identified structural relations may be modified for several reasons. Design rules may result in restructuring to obtain better modularity, encapsulation, extensibility and reusability, and mapping multiple inheritance to single inheritance.

---

[3] In the analysis/preparatory phase, the term objects may correspond to both classes or instances.

Object Interactions

As where the structural relations define the architecture of a system, the dynamic behaviour of the system is realised by object interactions, corresponding to the usage relations. Object interactions are represented by *message connections*[4]. A message connection simply indicates that two objects communicate. Message connections are usually identified after structural relations have been determined. In the design and implementation phases, object interactions may be modified as a result of various design decisions, for instance to improve reusability.

In general, software engineering principles such as encapsulation and modularity tend to collide with performance requirements. As a result, the interactions between objects may be modified during design and implementation phases to fulfil performance requirements.

In [Bergmans 91] important characteristics of object-oriented software development are described, and the problems that it attempts to tackle. In [Aksit 92b] a number of obstacles in object-oriented software development are addressed. A number of software development methods are described in [Wirfs-Brock 90b].

1.2.3 A Graphical Notation for Objects

In the subsequent chapters the structure of objects will be illustrated a number of times through a graphical presentation. The notation we use in these examples is in fact a subset of the notation that is described in detail in chapter 4. A brief explanation of this subset is shown here.

An object is depicted by a rectangle:

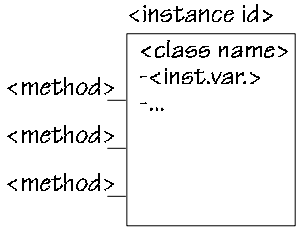


Figure 1.2.2 Graphical representation of an object.

Just inside the rectangle, at the top, is the name of the class, followed by a number of instance variable names, each preceded by a dash. If the specified object denotes a single instance, rather than a class, the name of the instance can be specified just above the rectangle. The figure shows a number of methods of the object, that are specified as labelled bars attached to the rectangle.

4 The terms *instance connections* and *collaborations* are used by some methods.

Three types of structural relations are discussed here: inheritance, delegation and part-of relations. The notation for these structural relations is defined as follows:

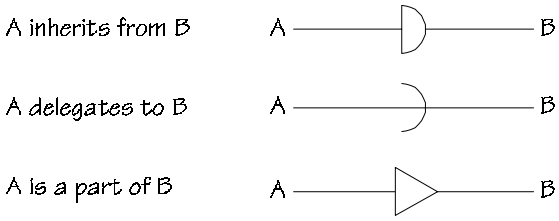


Figure 1.2.3 The notations for specifying the structural relations between objects.

As stated before, in chapter 4 a more detailed notation is described, but for the first part of this thesis the notation we described here is sufficient.

1.3 Thesis Outline

The contents of this thesis derive from the high-level software development trajectory as shown in figure 1.3.1; a problem specification is the input for the analysis and design phases, which eventually leads to the creation of programs. These programs are expressed in a particular language, with an underlying computation model. The execution of the programs requires tools like interpreters or compilers which map the high-level program specifications to a machine-understandable form (designated as 'executable' in the figure).

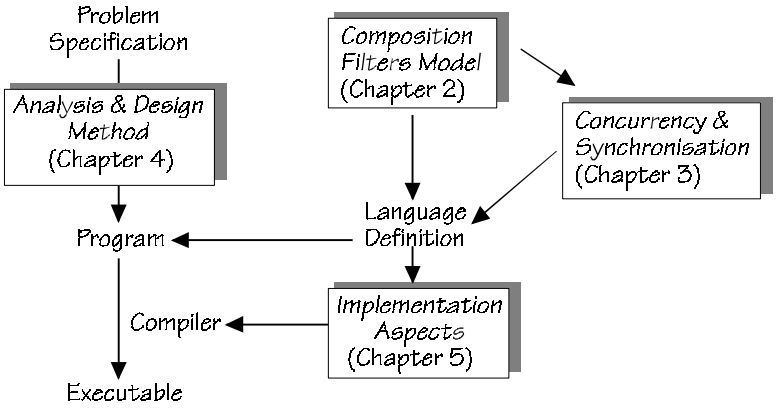


Figure 1.3.1 An overview how the chapters in this thesis relate to software development phases

In the forthcoming chapters the topic of concurrency and synchronisation is dealt with in relation to all these phases of software development. To start with, the underlying computation model is discussed. This is the basis that determines both the programming language and the analysis and design method. The reason is that the computation model determines the software engineering properties that we focus on.

The computation model is divided in two parts; one part is the composition-filters framework, and one part consists of the mechanisms for concurrency and synchronisation. The composition-filters framework is discussed elaborately, as it is the foundation of the work presented here, and should therefore be well-understood. This is the content of chapter 2; it explains the mechanism of composition filters and describes the most common filter types. Chapter two in fact describes background work as it does not introduce new research results.

In chapter 3 the topics of concurrency and synchronisation are introduced. In this chapter the integration of concurrency and synchronisation constructs in the composition-filters model is described. The prevalent issue is to fully support reusability and extensibility of objects in the presence of concurrency and synchronisation. Chapter 3 starts with an introduction to concurrency and synchronisation in object-based languages and systems. Subsequently an analysis is made of the interference between inheritance and synchronisation. This interference prohibits effective reuse and extension of concurrent objects, a phenomena that has been coined *inheritance anomaly*.

Section 3.3 explains how the creation of concurrency and the synchronisation of activities are addressed by the composition-filters model. The applicability of this approach is exemplified by a number of examples. The chapter concludes with an evaluation of the introduced mechanisms. The most important contribution of the proposed techniques are the reusability and extensibility properties of concurrent objects.

In chapter 4 the attention shifts from the modelling techniques to the issue of how to come up with an object-oriented representation of the entities in the application domain, and in particular how to come up with synchronisation specifications for objects. A simple analysis and design method is presented that is suitable for the development of composition-filters objects. The prime feature of the method is its support for the derivation of synchronisation constraints for objects. A graphical notation resembling state-transition diagrams is introduced that allows for the graphical specification and composition of synchronisation constraints. The notation can be translated algorithmically into composition-filters synchronisation specifications.

The implementation aspects of the concepts that have been introduced in chapter 3 are the topic of chapter 5. The three main topics that are addressed are respectively; reasoning about wait filters, optimisation of condition evaluations and the architecture of a framework that offers composition filter functionality. Chapter 6, finally, emphasises the contributions of the work and discusses future research.

The preferred order for reading chapters 2 to 5 is described by the following figure:

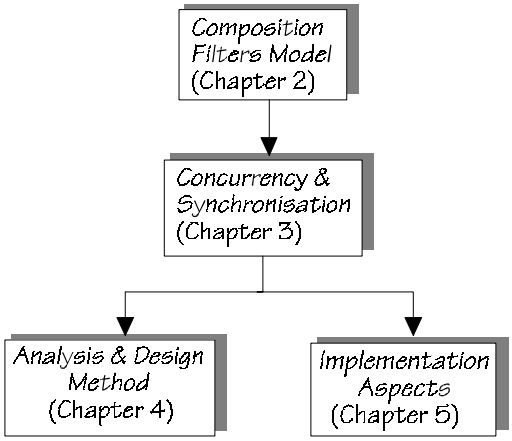


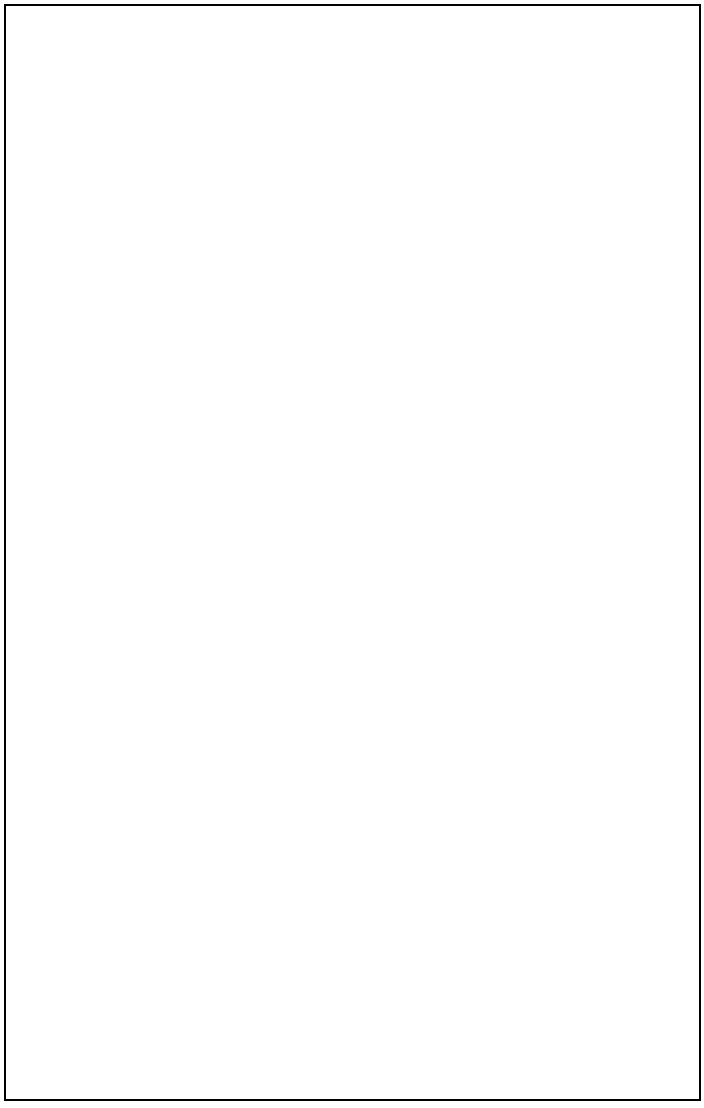
Figure 1.3.2 Dependencies between chapters.

The figure illustrates that chapter 2 is background work that contains necessary information to understand the other chapters. Readers that are very much familiar with the composition filters model may skip chapter 2. In chapter 3 the foundation of the thesis is laid; the key mechanisms are introduced here. These appear again in subsequent chapters. Chapter 4 and chapter 5 are independent from each other, and can be read in either order.

At a number of locations in this thesis we deemed a formal presentation of definitions and specifications appropriate. We adopted the METANOT notation [Meyer 90], which largely adheres to mathematical conventions, for these cases. This notation is briefly described in appendix A. In chapter 2 the programming language Sina is described. Appendix B specifies

the syntax of this language. In appendix C the interface description of some of the primitive classes of the Sina programming language is specified.

Chapter 2



The Composition-Filters Object Model

2.1 Introduction

In this chapter the computation model of the composition-filters object model is explained. The composition-filters model aims at providing a basis for the solution of a range of problems in the construction of very large and complex applications. It focuses mainly on modelling problems.

The composition-filters model is an evolution of the object model in early versions of the language Sina[1] as described in [Aksit 88], [Tripathi 88], [Aksit 89], [Tripathi 89] and [Aksit 91]. The term *interface predicate* was coined here as a mechanism for describing a range of data abstraction techniques. Further research in various application domains[2], including concurrency and synchronisation, stressed the need for additional constructs to be added to the language (see for instance [Aksit 92b] and [RICOT 94]). Instead of extending the language with numerous new language constructs, the framework of *composition-filters* was introduced which integrates all these desired constructs and the interface predicates into a single, unified model.

The presentation in this thesis first introduces the composition-filters framework, and subsequently discusses the incorporation of concurrency and synchronisation within this framework. Moreover, all the issues related to concurrency and synchronisation have been deliberately removed from the discussion in this chapter, to be left for chapter 3. A brief discussion of the concepts and applicability of composition-filters can be found in [Bergmans 92b].

The programming language Sina that is introduced in this chapter is an implementation of the composition-filters computation model. It plays the role of a research vehicle for demonstrating and verifying the concepts that are introduced in the composition-filters model. Since Sina is also used for the syntactical presentations, the distinction between the computation model and the language is often blurred. This should not cause serious problems, as Sina strictly adheres to all the concepts of the composition-filters computation model.

The Goals of the Composition-Filters Computation Model

The composition-filters model aims at providing techniques to support the construction of large-scale software (*Programming In the Very Large*, [Wegner 90]). One of the important aspects is the attempt to manage the complexity of such systems. The object-oriented model provides techniques to support this: strong encapsulation of modules promotes cohesion and reduces coupling. Polymorphism and inheritance[3] support effective reuse and

---

1 Named after the medieval philosopher, scientist and physician Ibni Sina (also known under the Latin name Avicenna).

2 It must be stressed that the development of the composition-filters framework has been a team effort of the TRESE project at the University of Twente as a whole, not of the author only.

3 Or similar reuse techniques such as delegation and composition.

extensibility of software, which in turn improves maintainability. The composition-filters computation model is based on the object-oriented model.

There are some other observations that influence our goals; firstly, various abstraction techniques can help in managing the complexity. Secondly, large-scale software incorporates multiple technical application-domains[4]. Thus we need a general-purpose computation model that offers high-level abstraction techniques.

The third observation is that, because of the large investments in terms of money, time, educating people, and equipment and the large life-time of these software systems, maintenance properties are of utmost importance. Thus we foresee great importance for evolving systems, demanding a high level of extensibility and reusability of software. This in particular requires sufficient modelling power to construct a system without infringing its extensibility and reusability properties.

> Even very simple and primitive languages, such as for instance assembly languages, can be used for constructing very large and complex systems. However, it is generally acknowledged that more modelling power is required to effectively cope with the problems involved in the construction of such systems. Examples of this are language constructs such as while-loops and if-then-else for managing the control flow, or abstraction mechanisms such as functions and abstract data types. What we would like to stress is that the ability to compute something is not the only issue involved in the design of a language or system. Software-engineering properties are very important as well.

Therefore, we adopt the following two requirements:

- Support multiple application domains in an extensible way. We want to cover these with a single framework, with consistent syntax and semantics. The model should provide a form of open-endedness to be able to cope with new application domains.
- A declarative[5] approach: this means that the externally visible behaviour of an object must be defined on the interface of the object, rather than being embedded in its implementation. Most library-based[6] approaches are based on explicit calls to library functions, merged in the application code. This is severely affects reusability and modularity.

The issue of performance has not been touched upon as yet, as it is not our first priority: if a certain expression exploits the very corners of flexibility and expressiveness of a model, this may lead to costly -in terms of efficiency- operations. We consider this to be acceptable. However, if an expression conforms to a very simple and straightforward concept, for instance one that is supported efficiently in other computation models and programming languages, it should be possible to recognise this situation, and come up with a likewise efficient implementation for that specific expression. Thus (during the optimisation phase) it

---

4 Examples of such domains are: distributed systems, databases, real-time systems, concurrency & synchronisation, etcetera. See for instance [RICOT 94], [Yücesoy 92] for a discussion on application domains.

5 Not in the sense of declarative programming languages such as Prolog.

6 The various application domains can be each addressed with a specific library, offering tailored functions or modules for the particular application domain.

should be guaranteed that 'simple' expressions have an efficient implementation, whereas complex and expensive expressions with inefficient implementations are acceptable.

> One of the arguments supporting this approach is that, if a complex expression is applied in a program, the application will require this. Thus, in a language that does not support this construct, the programmer will have to implement it by hand. Although this may be perfectly possible, it requires (a) additional implementation effort, (b) it is likely to be less efficient than a built-in construct and (c) the resulting program is very likely to have worse maintenance characteristics.

It should be clear, though, that we are interested in a model that has the potential to be effectively applied in practice. Therefore, we strictly want to avoid inherently inefficient constructs that offer no perspective of efficient realisation at all.

An Overview of the Composition-Filters Model

Before discussing the various aspects of the composition-filters computation model, we will briefly outline its concepts and components. The composition filters model is a modular extension to the conventional object model. The behaviour of an object, that is also referred to as the *kernel object*, can be modified and enhanced through the manipulation of incoming and outgoing messages only. To achieve this, the kernel object is surrounded by a layer called the *interface part*. The resulting model and its components are shown in the following figure:

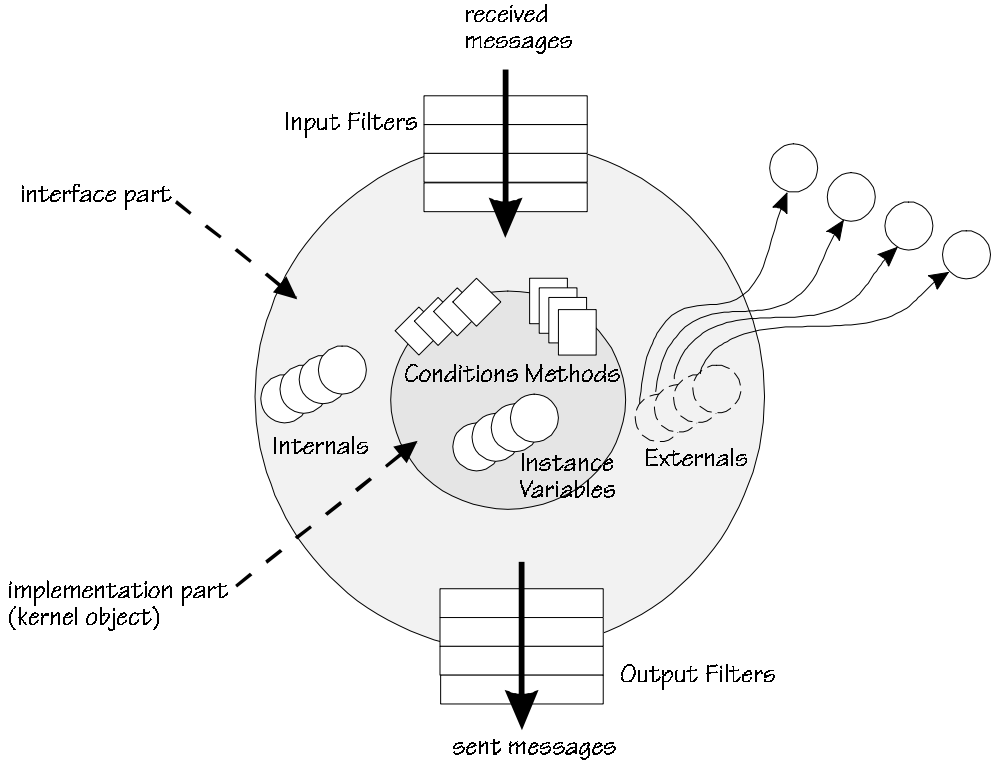


Figure 2.1.1 The components of the composition-filters model.

We will briefly discuss each of the components and the roles they play in the composition-filters computation model. The most significant components are the *input filters* and *output filters*. A single filter specifies a particular manipulation of messages. Various filter types are

available. The filters together compose the behaviour of the object, possibly in terms of other objects. These other objects can be either *internal* objects or *external* objects. The behaviour of the object is a composition of the behaviour of its internal and external objects, although these remain fully encapsulated.

In addition, -part of- the behaviour of the object will be implemented by the kernel object, which is therefore also referred to as the *implementation part*. The kernel object encapsulates a set of instance variables. On the interface of the kernel objects appear *methods* and *conditions*. The methods may be invoked through messages, assuming that the filters of the object allow this. Conditions are essentially boolean expressions that represent the state of the object. The conditions are used by the filters in deciding upon the manipulation of messages. As an example, a specific filter can reject messages, based on its properties or based on the value of a condition. All the components of the composition filters model will be explained in detail in this chapter.

On the Forthcoming Chapter

In the rest of this chapter, we will explain the composition-filters computation model, and the programming language Sina that supports this model. The emphasis is on concepts, rather than a detailed discussion of the language: this is not a language reference manual. Frequently, however, we do give the syntactical representations, as far as these are relevant, and when the explained constructs will be used in later chapters.

Not all -currently defined- aspects of the composition-filters model are discussed. In particular we omitted:

- Associativity, or 'queries', as we will not need it in the rest of this thesis and a discussion would complicate the explanation and formal model of filters. In [Aksit 92a] associativity in the composition-filters model is described.
- The support defined for some particular domains: the model provides a certain degree of open-endedness, which allows for adding support for new domains without affecting the semantics of the model. For example, the support for real-time constraints [Aksit 94b] is not discussed here.

The rest of this chapter is organised as follows: the next section describes the so-called kernel object model. In section 2.3 the principles and specification of composition filters are explained. Section 2.4 discusses how these are defined in Sina within the *interface part* of an object definition, and shows a number of applications, among which the basic data abstraction mechanism. In section 2.5 further aspects of the language, such as type-checking and scope rules are described. In 2.6 a precise definition of the basic computation model is given, the final section describes related work and discusses the characteristics of the composition-filters model.

2.2 The Kernel Object Model

The composition-filters model is an extension to the object-oriented model. Its object model can be presented as two separate partitions: one part is an -almost- conventional object-based model, which is surrounded by an encapsulating layer. The latter is the second part, containing the extensions that are made by the composition-filters model. The first part is the kernel of the object, and is called the *implementation part*, the second part is called the *interface part*, as it manages the incoming and outgoing messages.

The following figure shows the two layers, with some additional detail of the implementation part:

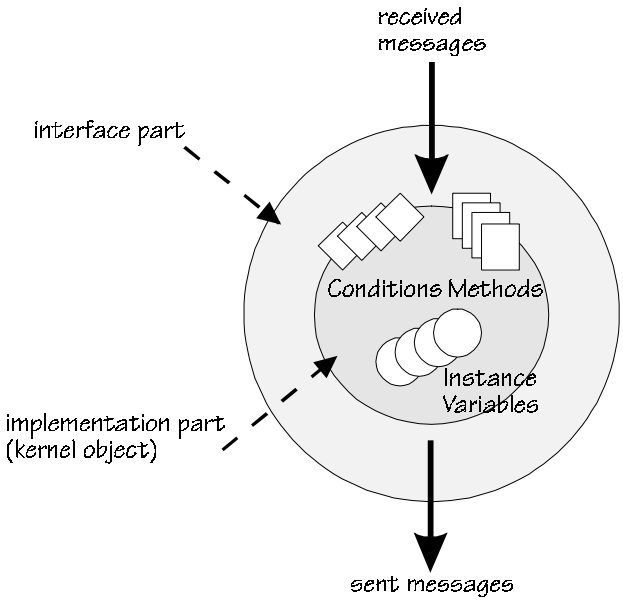


Figure 2.2.1 The two layers of the composition-filters object model.

The interface part receives the incoming messages, processes them in a manner that will be elaborately discussed later in this chapter, and then -optionally- hands them to the implementation part, where they lead to the execution of a method. This implementation part follows the conventional object-based computation model; it is called the *kernel object model*.

The implementation part can be replaced without severe consequences by virtually any conventional object-based or object-oriented language, such as Smalltalk [Goldberg 83], C++ ([Stroustrup 86], [Ellis 90]), CLOS [DeMichiel 87], Self [Ungar 87] or actor languages (e.g. [Agha 86], [Agha 88]). The discussion of the kernel object model in this section is specific to the language we use, Sina, and almost independent of the composition-filters model.

The composition-filters model is *class-based*: for an application model consisting of a number of objects, all objects with common characteristics are instances of the same class. Objects are only specified on the class level; this supports the creation of multiple instances,

and reuse through the inheritance mechanism. The now following discussion of object specifications deals with classes.

As we can observe in the preceding figure, the kernel object model contains the three main components *methods, conditions,* and *instance variables*. The names of methods and conditions are visible on the encapsulation boundary of the kernel object. They can be accessed from outside the kernel object, whereas this is impossible for the instance variables: these are fully encapsulated.

Instance Variables

Instance variables are declared as follows, where the left hand side declares one or more instance variable identifiers, and the right hand side defines a *type*:

```
instvars
    firstName, lastname : String;
    isMale : Boolean;
    birthDate : Date;
    home : Address;
```

All instance variables are first-class objects, we do not distinguish basic data types such as the types String and Boolean from user-defined object types (classes). The system provides the basic data types through a number of *primitive* classes. Examples are: Integer, Real, String and Boolean. Because the difference between a user-defined class and a primitive class is not visible to other objects in the system, the distinction will not be made explicit unless needed[1].

The right hand side of object declarations must always be an identifier that is associated with a class. From the specification of that class, a type definition is derived. The type in the instance variable declaration then serves two purposes: first, it expresses the type of the instance variable: all future assignments to the instance variable must obey the subtyping rules. Secondly, the type is used for the initialisation of the instance variable; when the object is created, for each instance variable an instance of the specified class is created.

Message Passing Semantics

One object can request a service from another object by sending it a message. Message passing between composition-filters objects follows a request-reply model: the client object[2] sends a message to the server object, requesting a service. The client object halts its execution until it receives a reply to the message from the server object. A message may include a number of parameters, where each parameter can be an arbitrary object. The server object is fully responsible for servicing the request, or eventually rejecting it.

In the general case, a message request will result in the execution of a method. When the execution is finished, a reply value is returned to the client object. The reply value is an

---

1 And in fact, it depends on the specific system implementation. New primitive types may even be introduced later to encapsulate low-level (system) features.

2 The notion of *client* objects and *server* objects is not meant to suggest the client-server paradigm for distributed computing. Rather, sending a message is considered a request to perform some service, therefore the notion of server (the object performing the service) and client (the object receiving the service) is adopted.

object. If no information needs to be returned, the reply can be a nil object. This is an instance of class Nil, and represents the most elementary object, with no internal state and only the minimal required behaviour. An explicit return statement in the method designates the result object. In absence of this statement, nil is returned.

Since the execution of a method body consists of message invocations as well, the execution of an object-oriented program results in a nested thread of message invocations, as illustrated in the following figure:

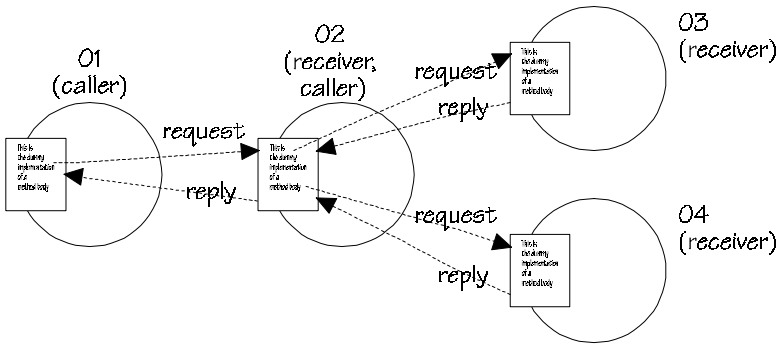


Figure 2.2.2 The chain of requests and replies between objects.

The subsequent message invocations form a thread of control that successively visits objects O1, O2, O3, O2, O4, O2 and back to O1 again.

Methods

The behaviour of an object is implemented by its methods: these define a number of actions that are performed in reaction to the invocation of the method. We refer to these actions as the *method body*. The method body consists of a message expressions and control structures. The full definition of a method consists of a method declaration, which specifies the name of the method, the names and types of the arguments and the return type. We illustrate this with an example of a complete method definition:

```
methods
    getFullName(order : Boolean) returns String
    comment "compose a full name from the first and the last name, when <order> is true,
            the normal order (firstName+lastname) is used, otherwise this is reversed"
    temps
        fullName : String;
    begin
        if order
        then fullName := firstName.cat(' ').cat(lastName)
        else fullName := lastName.cat(', ').cat(firstName) ;
        return fullName
    end ;
```

This code example defines a method getFullName, with a single, Boolean argument, order. The method must always return an object of type String. A method definition includes a comment clause, which is intended to describe the meaning and responsibilities of the

method[3]. After the method declaration, the temporary objects that are used in the method implementation are declared. In this example the string variable fullName is the only temporary object declared for the object.

The body of method getFullName consists of two statements; an *if-then-else* statement[4] that chooses among the two possible representations and assigns the properly formatted string to the temporary variable. The cat message is used to concatenate strings. The last statement is a *return* statement that returns the value of the fullname variable.

Conditions

Conditions are a special type of methods; a condition is a method that takes no parameters and returns a Boolean result. The purpose of conditions is to provide information about the current state of the object (i.e. the object *is*, or *is not* in a particular state). This is similar to guards in for example Procol [Bos 89] and ABCL [Matsuoka 93b]. Conditions should not have any side-effects, as they may be evaluated repeatedly and in a non-deterministic way. Conditions are primarily intended to be used in the interface part of the object, but may be referred to within the implementation of methods or other conditions.

Two examples of conditions:

```
conditions
    IsBirthday
        comment "compares the current date, stored in the global <Calendar>, with the
                 birth date"
        begin
            return (birthDate.day=Calendar.day) and (birthDate.month=Calendar.month)
        end;
    IsLocal
        comment "compare the city of the home address with <ThisCity> "
        begin
            return home.city=ThisCity
        end;
```

These examples show that the conditions are expressed by arbitrary message expressions. Usually just a single expression is sufficient, but the use of multiple statements and temporary variables is allowed.

The restrictions that apply to the implementation of conditions (especially side-effects and returning a Boolean value) cannot be fully checked at compile time, due to the dynamic binding in object-oriented languages. For instance, the expression 'home.city' in the implementation of the isLocal condition above calls the city method on the home instance variable. Because in general, any object that satisfies the subtyping rules can be assigned to home during the

---

3 Apart from these comment clauses, lexical comment brackets '/* */' and '//' are supported as well. The explicit comment clauses can be exploited by a programming environment because they carry additional semantic information: it is known what entity the comment describes.

4 We do not go into the details of the syntax, such as the fact that in some implementations of the language all control structures are translated by a pre-processor into message expressions on blocks (i.e. first-class method bodies), similar to the approach in Smalltalk-80 [Goldberg 83]. Common operators such as for performing arithmetic and relational operations are translated to message expression as well by a pre-processor.

execution of the program, it cannot be checked at compile time whether the city method has any side effects. Similar arguments apply to the fact whether the returned result really is a Boolean object. We conceive three approaches to this problem: the first is by run-time checks, the second is through extended subtyping rules and the third is to make it the responsibility of the developer.

- Returning a Boolean object as the result of the condition is a generic type-checking problem, and can be solved as such. Note that type-checking is partially performed at run-time, again because of dynamic binding[5]. A compiler can insert code for run-time checks on side-effects, however, this is likely to bring substantial performance loss. In addition, this is a 'late' solution that can only indicate conflicts when a violation is detected.
- A more secure approach is by enforcing strict sub-typing rules. The sub-typing rules are then to be extended with the rule that an object Sub is only a subtype of another object Sup if for all methods of Sup that are free of side-effects, Sub provides corresponding side-effect free methods. Although a minor part of the sub-type checks are performed at run-time, this allows for a largely static verification of the constraints. The disadvantage of this approach is that the new subtyping rule imposes a strong restriction[6]. We feel that this is a too strong limitation on the expressiveness of the language.
- The last alternative is to make the developer responsible for avoiding side-effects in condition expressions. The main motivation for this is that no satisfactory compile-time techniques can be defined without severe limitations. This is mainly due to the dynamic binding in object-oriented programs.

For Sina, the third approach was adopted, which leaves the responsibility to the developer. Potential problems can be avoided through methodological support and by making the developer aware of the problems.

Initial Method

The final property of the kernel object model that is discussed here is the so-called *initial method*. This is a dedicated method body, specified optionally, which takes care of the initialisation of the object: when a new instance is created, first the initial method is executed, before any other method. For example:

```
initial
    begin birthDate := Calendar.date end;
```

This initial method assigns the current date to the birthDate instance variable when the object is created. The initial method is in general used to initialise instance variables, and may initiate other activities in the system.

---

5 In a closed system (an application with a fixed number classes) for each expression a set of potential classes can be determined [Palsberg 91]. Based on this information possible violations can be detected. However, this is a pessimistic approach that will not approve a large share of correct condition implementations, and in addition can only be applied in closed systems.

6 We should note, however, that this approach can be very advantageous for optimisation purposes, e.g. for an efficient implementation of synchronisation constraints, or for replication in a distributed system.

An Example of a Class Definition

We combine the examples given above into a single class specification. This specification is the implementation part of a composition-filters class definition. The class is labelled 'Person', a separate comment clause is available to describe the responsibilities and properties of the class:

```
class Person implementation
    comment "This is the implementation of class Person; it stores the common properties
            of all person objects in the system";
    instvars
        firstName, lastname : String;
        isMale : Boolean;
        birthDate : Date;
        home : Address;
    conditions
        isBirthday
            comment "compares the current date, stored in the global <Calendar>, with the
                    birth date"
            begin return (birthDate.day=Calendar.day) and
                    (birthDate.month=Calendar.month)  end;
        isLocal
            comment "compare the city of the home address with <ThisCity> "
            begin return home.city=ThisCity end;
    initial
        begin  birthDate := Calendar.date  end;
    methods
        getFullName(order : Boolean) returns String
            comment "compose a full name from the first and the last name, when <order> is
                    true, the normal order (firstName+lastname) is used, otherwise this is
                    reversed"
            temps
                fullName : String;
            begin
                if order
                then fullName := firstName.cat(' ').cat(lastName)
                else fullName := lastName.cat(', ').cat(firstName) ;
                return fullName
            end ;
end // class Person implementation
```

The last line of the class definition demonstrates another way of writing comments in the code: the '//' symbol, following the C++ conventions, designates the start of a comment that takes up the rest of that line (and no more).

After the description of the kernel object model, the following section will describe the extensions provided by the composition-filters model.

2.3 Composition Filters

2.3.1 The Extended Composition-Filters Object Model

The composition-filters object model extends the kernel object model with a layer that is called the interface part. The major aspect of this layer are the *input filters* and *output filters*. Input filters deal with the messages that are received by the object, whereas output filters deal with the messages that are sent by the object. The following illustration depicts the object model including the filters:

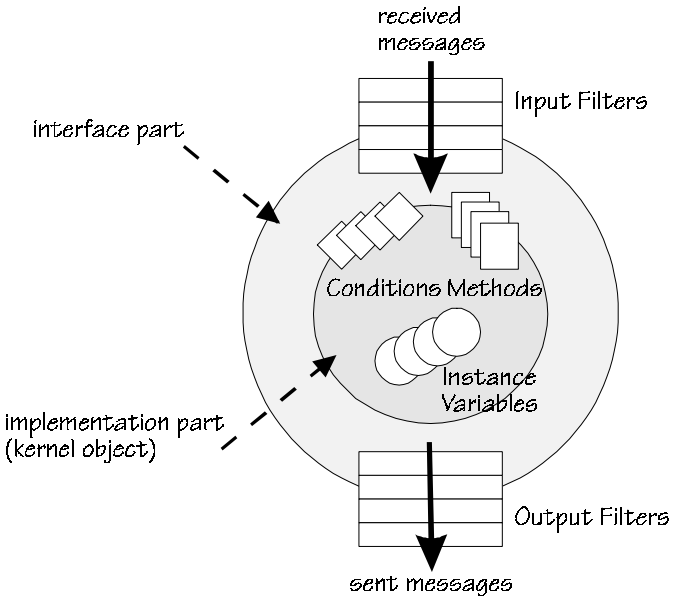


Figure 2.3.1 The composition-filters object model with input and output filters.

The filters are the crucial aspect of the composition-filters paradigm: because all activities in an object-oriented computation model are performed through message invocations, the manipulation of messages can control virtually every aspect of object-oriented computation. The composition-filters model applies a form of reflection on messages to be able to adapt to the varying constraints and requirements imposed by different application domains. See for instance [Ferber 89] for a discussion on the various approaches to reflection in object-oriented languages.

An important distinction with most approaches to reflection is that the composition-filters model provides *limited reflection* on messages; it intends to provide a level of abstraction that combines expressive power with befitting software engineering properties. It supports the construction of complex applications by providing a range of domain-specific techniques in a single framework, such that every solution can be specified in a consistent manner.

In this section the syntax and semantics of composition filters are discussed, the other components of the interface part will be introduced in the subsequent section.

2.3.2 The Principles of Composition Filters

We will explain the basic mechanism of composition filters with the aid of figure 2.3.2, that is shown below. The discussion focuses on input filters, but the output filters are described in exactly the same manner. The main difference is that output filters deal with sent instead of received messages.

To understand the schema the following should be kept in mind: filters are defined in an ordered set. A message that is received will pass the filters in the set, until it is discarded or can be dispatched[1]. In this chapter we do not bother with the issues related to concurrent activities and multiple -simultaneous- message invocations on an object.

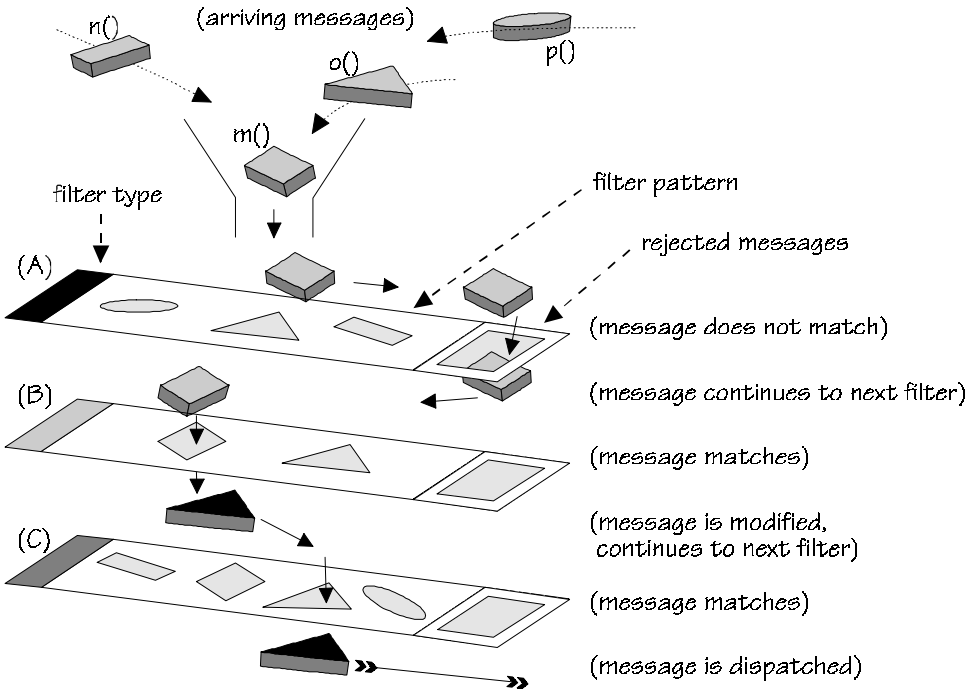


Figure 2.3.2 A schematic representation of the filter mechanism.

The figure visualises the processing of messages by three filters, A, B and C. An object can receive a variety of messages, in the figure respectively labelled m(), n(), o() and p(). All received messages are subject to manipulation by the successive filters. Different types of filter exist for different manipulations on messages. We follow the message m() as it passes through the filters.

Each filter specifies a pattern that a message is compared to. The pattern is defined by a syntax that is independent of the specific filter type. It can be defined in terms of message properties, but the matching may also depend on the current state of the object. In the figure, message m() does not match with the pattern defined by filter (A). Thus, the message is *rejected* by the filter. All messages that are rejected by the filter are treated in a way that depends on the specific type of the filter.

---

1 With *dispatching* we mean that the message is activated again, for example to start executing a method body, or to be offered at the interface of another object.

In the example, the rejected message is simply passed on to the next filter. The pattern that is defined by filter (B) matches with the message. This is referred to as *acceptance* of the message by the filter. This initiates a particular action, that depends on the filter type. The message may be manipulated and modified, and the filter type also determines what happens next to the message. In the example of filter (B), the message is modified (designated in the figure by its changed shape and colour), and then passed on to the next filter.

For the last filter in the example, filter (C), the pattern also matches the message. The acceptance of the message again implies an action to be performed that depends on the filter type. In this case, this causes the message to be dispatched, for example to a local method of the object. The message itself contains the information as to where it should be dispatched (i.e. the target object and the message selector), but this information can be manipulated by the filters.

In general, every filter set contains a filter that causes messages to be dispatched, as this is the only means to trigger the execution of a method. For output filters, dispatching means that the message is sent to the target object. Note that upon its reception by the target object, the message must first pass the input filters of the target object. Except for the functionality of message dispatching, filter types may define actions that generate exceptions, or modify certain properties of the message.

In summary, a filter consists of a pattern definition and a filter type. Messages are matched against the pattern, then the filter type determines the action to be performed upon acceptance, respectively rejection. Next we will discuss the mechanism for matching a message in a filter and the definition of the matching pattern with more detail.

2.3.3 The Filter Mechanism

The filter mechanism aims at providing a generic notation for matching and modification of the two most important message properties. These properties are the *selector* and the *target* of the message. The selector is the label of the message, requesting a specific service, and the target is the object that is supposed to implement this service. Together, the target and the selector designate a specific method implementation[2]. To match on other properties of messages, or select messages based on the state of the system, *conditions* are to be used. The modification of message properties other then the selector and target is the responsibility of specific filter types. Conditions and filter types are discussed in detail later in this section.

In figure 2.3.3 the filter mechanism for matching and modification of messages is illustrated. We refer to the modification of the target and selector of a message as *target substitution* respectively *selector substitution*. The figure demonstrates the processing of a message m by four successive filters A to D. A received message has to pass through all the filters to result in a successful dispatch.

---

[2] This is precisely what distinguishes message sending from a conventional function call: different targets may have a different implementation for the same message selector, thus realising polymorphism.

Each filter consists of an ordered set of *filter elements*. Each filter element is a separate message processing entity that can perform matching and substitution. The message is applied to each of the filter elements, from left to right, until a match has occurred. A substitution will only be performed by a filter element when the message has first matched at the same filter element.

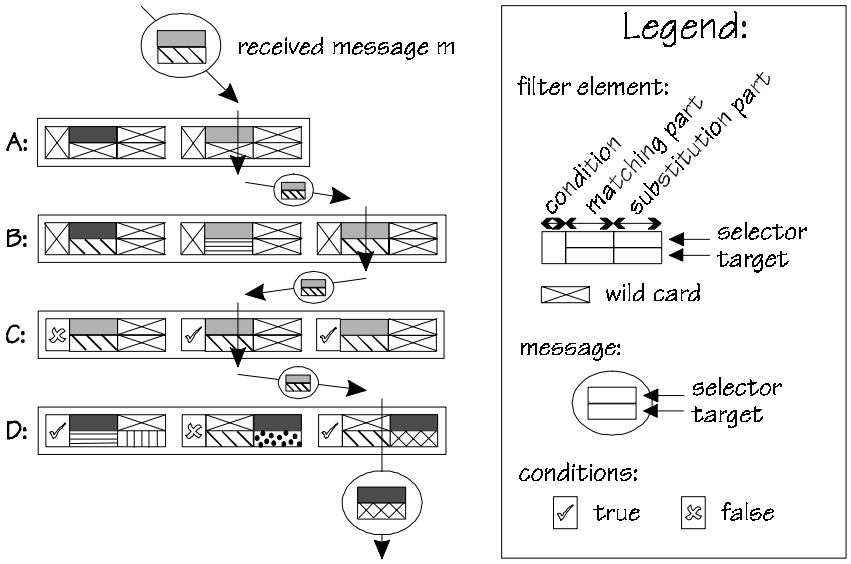


Figure 2.3.3 A visual representation of filter specifications

A filter element consists of three parts:

- A *condition*, which specifies a necessary condition to be fulfilled in order to continue the evaluation of a filter element.
- A *matching part*, in which the evaluated message is matched against a pattern consisting of a message selector and a target specification.
- A *substitution part*, which specifies replacement values for the target and selector of the message. If the substitution part is a wild card, the target and selector remain unchanged.

We explain the filter mechanism through the concrete example in figure 2.3.3. In filter A, the selector of the received message is matched against the selector of the matching part of each filter element; when the filter element does not match the subsequent filter element is tried. In filter A only the second element matches the message since the selector specified by the first filter element differs from the selector of message m. When a message matches one or more of the filter elements, the message is accepted by filter A as a whole.

In filter B, matching is not restricted to the selector of the message, but involves the target of the message as well. The first element of B does not match (selector differs), neither does the second filter element (target differs) but the third element does match. The message proceeds, unchanged, to the third filter. It should be noted that upon reception of a message by the object, the target of that message is always the object that just received it. Only after modification of the target in one of the filters the target may be different from *self* (this is a pseudo-variable that always refers to the current object).

Filter C differs from the previous filters in that each filter element specifies a condition, which can be either *true* or *false*. A condition can be considered as a constraint or precondition on filter elements: only when the condition is *true*, the filter element will be evaluated, otherwise it is skipped. A condition is specified by an expression of arbitrary complexity. In the previous section, we described how conditions are implemented in the implementation part of an object. In the figure we only show the resulting Boolean value. In filter C, the message would match with the first filter element, but this is skipped because its condition is *false*. In the second filter element, both the condition is satisfied and the message matches, and thus the filter accepts the message.

Filter D demonstrates all components of filter elements. It shows substitution of selectors and targets. When both the condition of a filter element is satisfied and the message matches, a new target and/or selector can be substituted. In filter D, the first filter element does not match and the second filter element has a condition that is *false*. The message is accepted at the third element and new values for the target and selector of the message are substituted, unless the filter type designates otherwise.

The type of the filter determines what will happen with the message. Commonly the last filter dispatches the message to destination that is specified by the -updated- combination of target and selector.

The conditions, the matching and the substitution that are provided by filters form a generic mechanism for selecting messages based either on their properties (selector or target), or on some condition specified by the receiving object. They also support the renaming of messages (substitution of selectors) and redirection of messages (by substituting new targets). Based on the acceptance or rejection of a message, the filter can perform appropriate actions such as bouncing a rejected message or dispatching an accepted message.

2.3.4 Syntax of Filter Specifications

Now that the matching and substitution of filters has been explained, we will describe how they are specified. The specification of filters in a Sina program is in fact the declaration and initialisation of a filter object. As any declaration, this consists of a filter label, a filter type, and an initialisation. An ordered set of filters is defined through a number of successive filter declarations, the declaration order defines the ordering of the filters in the set.

> The reason for making filters first-class objects is to prepare our model for the future: the representation of filters as objects maintains a unified presentation for entities in our model. This allows for accessing and manipulating filters, just as any other object. In addition, as a filter is an instantiation of a class, new filter classes can be defined. These may substantially extend the expression power of the object model, without modifying the model that is offered to the programmer. This modular extension of the semantics of the object model makes the composition filters particularly useful as a research vehicle. But it can also be significant for a practical application of the model in industrial software development environments.
>
> Because a filter is a first-class object, there is no theoretical objection to let programmers define their own filter class, and creating instantiations of such a class. This would implement full

reflection on message acceptance and message sending, providing a very powerful tool[3]. However, we consider this as an inappropriate approach, as it does not sufficiently help in solving the problem of how to manage complexity; abstraction techniques are still needed. Instead we propose *limited reflection* where domain experts can provide powerful and expressive tools (i.e. filter classes) to express domain-specific functionality, within a rigid framework of abstraction techniques.

One might conceive problems with infinite recursion because objects have filters which are objects themselves and in turn may have filters, etcetera. This would indeed be the case if all objects and filter definitions were user-defined classes. However, because the system provides a number of built-in primitive classes and filter types, this boot-strapping problem can be solved. In fact, the functionality of some filter classes can only be defined at this lower implementation level, as they are too much interwoven with the object semantics.

Consider the following example of two syntactic filter declarations for the class Person of which the implementation part was defined in the previous section:

```
select : Error = { True=>[*.getFullName]*.*, IsBirthday=>[*.getPresent]*.* };
disp : Dispatch = { inner.* };
```

On the left hand side of the "=" symbol we find ordinary object declarations; each filter object has a label and a -filter- type, or -filter- class. In this example the first filter is of type Error. An error filter has very straightforward semantics: when a message matches, it can just pass to the next filter (provided this is available). When a message cannot match with any of the filter elements, an error exception is raised, and the execution is halted. Thus, an error filter is used to reject unwelcome or unacceptable messages.

The expression between curly brackets is the *filter initialisation* clause. This is the specification of a number of filter elements that form the initialisation of the filter object. The select filter consists of two filter elements, separated by commas. We examine the first element in detail:

```
True=>[*.getFullName]*.*
```

This filter element consists of the three parts: a condition, a matching part and a substitution part. The condition part, "True", is followed by the *enable operator* "=>". The condition True is available for all objects. As the name suggests, it is always valid. The following two definitions describe the semantics of the condition and the enable operator:

Definition 2.3.1: Conditions

A condition *C* that is part of a filter element *FE*, causes the *FE* to be rejected when *C* is not valid (i.e. has the Boolean value *false*).

Definition 2.3.2: The* enable *and the* exclusion *operators

A filter element *C=>M*, with the enable operator "=>" defines that, if *C* is valid, all messages that match with *M* are accepted, and substitutions are applied to the message. The messages that do not match with *M* are rejected.

A filter element *C~>M*, with exclusion operator "~>" defines that, if *C* is valid, all messages are accepted, except those that match *M*. Optional substitutions defined by *M* are discarded.

---

3 This is in fact very similar to the model proposed by the MAUD language [Frølund 93].

On the right hand side of the enable operator we find, between square brackets "[" and "]", the matching part, in this case "*.getFullName". The matching part consists of a target specification and a selector specification, separated by a dot. The target specification is either an identifier of an object, or the *wild card symbol* "*". The wild card symbol, which can be applied for both target and selector specifications, has the function of a don't care: any target or selector will match with it. This means that the matching part will match on all messages with selector getFullName, regardless of the target of the message.

The substitution part follows the matching part, and in the select filter it is defined as "*.*". This means that for both the target and the selector a wild card is specified, which does not modify the current values of the message. To summarise the meaning of the first filter element, we can suffice by saying that it accepts all messages with a selector getFullName, and rejects all others.

The second filter element, "IsBirthday=>[*.getPresent]*.*", is almost the same as the previous one, but it matches on message selector getPresent, and instead of the condition True, it has the condition IsBirthDay associated with it. This condition was defined in the previous section as:

```
IsBirthday
    comment "compares the current date, stored in the global <Calendar>, with the birth
date"
    begin return (birthDate.day=Calendar.day) and (birthDate.month=Calendar.month) end;
```

The most significant aspect of this condition is that it depends on the state of the object and it is obvious that this may change during the lifetime of the object. Thus, whenever a getPresent message is received, depending on the current value of the instance variable birthDate, and of the state of the Calendar object, the message is either accepted or rejected by this filter element.

Since the getPresent message cannot be accepted by another filter element, the condition IsBirthday directly determines whether the select filter accepts or rejects the message. When the condition is *true*, the message will be accepted by the error filter, and it may continue to the next filter in the set. But when the condition is *false*, the getPresent message will be rejected by the filter, thus raising an error.

The second example filter (disp) demonstrates some simplifications that are allowed in filter expressions and an extension to the filter semantics described so far: *signature-based target substitution*. The filter is of type Dispatch, and is initialised with a single filter element, "inner.*". The filter element does not specify a condition, has no enable or exclusion operator, nor a matching part.

For unspecified conditions and operators the following filter simplification rules apply:

Definition 2.3.3: Default condition

If no condition is specified by a filter element, the default condition *True*, which is always valid, is assumed.

Definition 2.3.4: Default enable operator

If a filter element does not specify an enable or exclusion operator, by default the enable operator is substituted.

Through application of these rules it can be shown that the disp filter initialisation above is equivalent to "{ True=>inner.* }". Because of the missing matching part (between square brackets) default matching rules apply:

Definition 2.3.5: Absence of Matching Part

If the matching part of a filter element is not specified, the following two rules define the matching criterion:
(a) The selector of the message must match the substitution selector.
(b) The signature of the substitution target must include the selector of the message.

Because the substitution selector must match the message selector, the latter will always be substituted with the same value, having no net effect. The target of the message may be replaced with a new value, though. Each object has a *signature*, which is formed by the set of all messages that an object may accept during its life-time[4].

The target inner in the filter element "inner.*" is a pseudo-variable that refers to the kernel of the current object. Thus, in its signature are only the methods that are implemented by the object itself. As a result, this filter element will match only for all the messages that conform to local methods of the object.

The *Dispatch* filter type causes all accepted messages to be delegated to the -possibly substituted- targets. If the target equals inner, this is the kernel object, that will simply execute the method with the same label as the message selector. When the target is another object then the message will be redirected to the target object, using the delegation mechanism [Lieberman 86]. The effects and applications of the dispatch filter type will be discussed in detail later in this chapter.

The two filters select and disp describe the following interface behaviour for class Person: only two methods are accepted, getFullName and getPresent, the latter only under certain conditions. These two messages are dispatched to the local methods of the class with the same name (assuming these methods are indeed defined).

A few additional rules for filter specifications are provided, mainly for convenient filter specification:

Definition 2.3.6: Missing target specifications

The target specifications in both the matching part and the substitution part are optional. When these are not defined, the dot between target and selector specification must be omitted as well. The default target specification that is assumed in these cases, is the wild card symbol.

Wild cards always match the target or selector in the matching part, and do not modify the target or selector when specified in the substitution part. We can apply definitions 2.3.3, 2.3.4 and 2.3.6 to the select filter specification, to simplify the filter initialisation clause:

```
select : Error = { [getFullName]*, IsBirthday=>[getPresent]* };
```

---

4 In the context of type checking the signature includes for each message the types of its arguments and its return type.

However, when considering definition 2.3.5 that defines the semantics when no matching part is specified, it appears that a filter element "[getFullName]*" is equivalent to the filter element "getFullName". Thus, we can suffice with the following filter specification:

```
select : Error = { getFullName, IsBirthday=>getPresent };
```

Finally, we will address two more syntactic extensions to the notation for filter specifications. The first allows the definition of multiple, alternative, conditions for a single filter element, the second allows the definition of multiple matching and substitution patterns, henceforth called *message processors*, in a single filter element:

Definition 2.3.7: Alternative conditions

A filter element with multiple conditions "*{C1, C2, .., Cn}=>FE*" is equivalent to multiple filter elements "*C1=>FE, C2=>FE, ... , Cn=>FE*". The conditions *C1*, *C2*, .., *Cn* are alternatives; when one of these conditions is satisfied, the filter element will be further processed.

An example of multiple conditions would be:

```
select : Error = { getFullName, { IsBirthday, isXmas }=>getPresent };
```

In this filter the getPresent message is accepted when either the isBirthday or the isXmas condition is valid.

Definition 2.3.8: Multiple message processors

The message processor may be replaced with a set of message processors between curly brackets; a message then matches if it can match one or more of the message processors in the list.

An example of multiple message processors:

```
disp : Dispatch = { True=>{inner.getFullname, inner.getPresent} };
```

In this example we have defined separated message processors for each of the messages that is accepted by the select filter. For illustrative purposes we added the default condition and enable operator.

Note that a filter element with an enable operator and multiple message processors can be split into separate filter elements, but that this is not possible in case the exclusion operator is specified.

2.3.5 A Formal Description of Message Processors

The precise semantics of the filter mechanism is given in section 2.6. Here we describe the syntax and semantics of the message processors, to give a precise definition of the matching and substitution rules. We conclude with a table that provides instant semantics for all combinations of identifiers and wild cards.

The slightly complicated, and in part redundant semantics for filter element specifications are partly due to historical reasons, and partly due to an effort to make the filter specifications effective from a software engineering point-of-view. The current notation for specifying composition filters is compatible with the notation adopted by *interface predicates* (e.g. in [Aksit 88] and [Aksit 89]). But a more important design issue was to provide a notation such that much used and intuitive simple patterns can be specified in a simple, straight-forward way. The previous filter example demonstrates this: matching purely on a message selector indeed requires specifying only the message selector.

The syntax of message processors is defined as follows:

<*filterElement*> $\stackrel{def}{=}$ (<*matchingPart*>, <*substitutionPart*>).
<*matchingPart*> $\stackrel{def}{=}$ '[' , (<*matchTar*>, '.')OPT, <*matchSel*>, ']'.
<*substitutionPart*> $\stackrel{def}{=}$ (<*substTar*>, '.')OPT, <*substSel*>.
<*matchTar*> $\stackrel{def}{=}$ '*' | <*Identifier*>.
<*matchSel*> $\stackrel{def}{=}$ '*' | <*Identifier*>.
<*substTar*> $\stackrel{def}{=}$ '*' | <*Identifier*>.
<*substSel*> $\stackrel{def}{=}$ '*' | <*Identifier*>.

The following definition specifies an algorithm in pseudo-code that gives a precise definition of matching and substitution in a message processor.

Definition 2.3.9: The semantics of message processors

A message *mess*, with a selector *mess.selector* and a target *mess.receiver* that is processed by a message processor of the form "[*matchTar.matchsel*]*substTar.substSel*" will result in a new message and a Boolean value that designates whether the message matched. The values of the results are defined by:

MessageProcessor(*matchTar, matchSel, substTar, substSel, mess*):<*Message, Boolean*> $\stackrel{def}{=}$
if *matchingPartUndefined* (1)
then if (*substSel*='*' ∨ *substSel*=*mess.selector*) (2)
∧ (*substTar*='*' ∨ *mess.selector*∈*sig*(*substTar*)) (3)
then if *substTar*≠'*' **then** *mess.receiver* := *substTar* **end**; <*mess, true*> (4)
else <*mess, false*> **end** (5)
else if (*matchTar*='*' ∨ *matchTar*=*mess.receiver*) (6)
∧ (*matchSel*='*' ∨ *matchSel*=*mess.selector*) (7)
then if *substTar*≠'*' **then** *mess.receiver* := *substTar* **end**; (8)
if *substSel*≠'*' **then** *mess.selector* := *substSel* **end**; (9)
<*mess, true*> (10)
else <*mess, false*> **end** (11)
end (12)

The algorithm is divided in two parts: the first part, in lines 2-5, deals with signature-based matching, and the second part, in lines 6-11, deals with complete matching and substitution rules. The input consists of a matching target and selector, a substitution target and selector, and the message that is to be processed. The components are either undefined, the wild card symbol '*', or an identifier[5]. The relevant components of the message are its receiver (cf. target), and the message selector. These can be accessed through the expression "mess.receiver" and "mess.selector", respectively.

In line 1 the distinction whether a matching part is defined, or not, is made. Undefined targets are replaced by their default value: a wild card. Whenever matching of targets or selectors is done, the presence of a wild card will always result in a match. In addition, a wild card never substitutes a selector or target.

---

[5] We do not go into the implementation details of matching; this could be based on string comparison, as is suggested in this section, but this is not necessary. For example, in an implementation, target matching could well be done through comparison of object pointers.

If no matching part is defined, matching will be based on the signature of the target. This is done in lines 2 and 3: in line 2 the substitution selector is compared with the message selector and in line 3 the signature of the substitution target (designated as "*sig(substTar)*") is compared with the message selector. If both the matching selector and the signature of the target match the message, the message is accepted. If this is applicable (when the substitution target is not a wild card) a new target is assigned to the message. The modified message and the Boolean value *true* are the result. If the message did not match, the unchanged message and the value *false* are returned.

In the case that a matching part is specified, the specification in lines 6 to 11 is followed. The matching condition is that both the target and the selector must match, where a wild card always matches (lines 6 and 7). When the match succeeds, both the target and the selector are substituted. For each of these substitutions the rule applies that in case of a wild card, no substitution is made. The modified message together with the value *true* is returned, and when the match failed, the -unmodified- message together with the Boolean value *false* is returned.

nr.	a	b	c	d	Preconditions	Tar.	Sel.	Explanation
1	a	b	c	d	tar=a ∧ sel=b	c	d	only if <a,b>, substitute <c,d>
2	a	b	c	*	tar=a ∧ sel=b	c	-	only if <a,b>, tar:=c
3	a	b	*	d	tar=a ∧ sel=b	-	d	only if <a,b>, selector:=d
4	a	b	*	*	tar=a ∧ sel=b	-	-	only if <a,b>, just match
5	a	*	c	d	tar=a	c	d	if tar=a then substitute <c,d>
6	a	*	c	*	tar=a	c	-	if tar=a then tar:=c
7	a	*	*	d	tar=a	-	d	if tar=a then selector:=d
8	a	*	*	*	tar=a	-	-	if tar=a then just match
9	*	b	c	d	sel=b	c	d	if selector=b, substitute <c,d>
10	*	b	c	*	sel=b	c	-	if selector=b, tar:=c
11	*	b	*	d	sel=b	-	d	if selector=b, selector:=d
12	*	b	*	*	sel=b	-	-	if selector=b, just match
13	*	*	c	d	true	c	d	always substitute <c,d>
14	*	*	c	*	true	c	-	always substitute target c
15	*	*	*	d	true	-	d	always substitute selector d
16	*	*	*	*	true	-	-	don't change the message
17	⊥	⊥	c	d	(sel=d) ∈ sig(c)	c	-	if selector=d, and d in signature of c.[6]
18	⊥	⊥	c	*	sel ∈ sig(c)	c	-	try to match signature of c.
19	⊥	⊥	*	d	sel=d	-	-	match selector only, since ∀d:d∈sig('*')
20	⊥	⊥	*	*	true	-	-	always match

6 If d is not in the signature of c, then this expression can never match.

The preceding table enumerates most of the possible variations in message processor specifications. Rows 1 to 16 show all combinations of identifiers and wild cards, and rows 17 to 20 show the signature based target substitution that is adopted when no matching part is defined.

The table adopts on the following definitions:

1. The message processor is defined as "*[a.b]c.d*":
   a $\stackrel{def}{=}$ matching target
   b $\stackrel{def}{=}$ matching selector
   c $\stackrel{def}{=}$ substitution target
   d $\stackrel{def}{=}$ substitution selector
2. The message has a receiver "*tar*" and a selector "*sel*".
3. The signature of a target *c* is designated as "*sig(c)*".

The columns labelled *a*, *b*, *c*, and *d* define the message processor. The column *preconditions* defines the conditions for message acceptance and matching. The substitution values defined by the columns "*Tar.*" and "*Sel.*" are only substituted when the preconditions are satisfied.

2.4 The Interface Part

The previous section explained the mechanism of composition filters, and described its semantics in detail. In this section it is explained how composition filters fit in the object model and how they can be applied.

2.4.1 The Components of the Interface Part

The following figure shows the components of the object model:

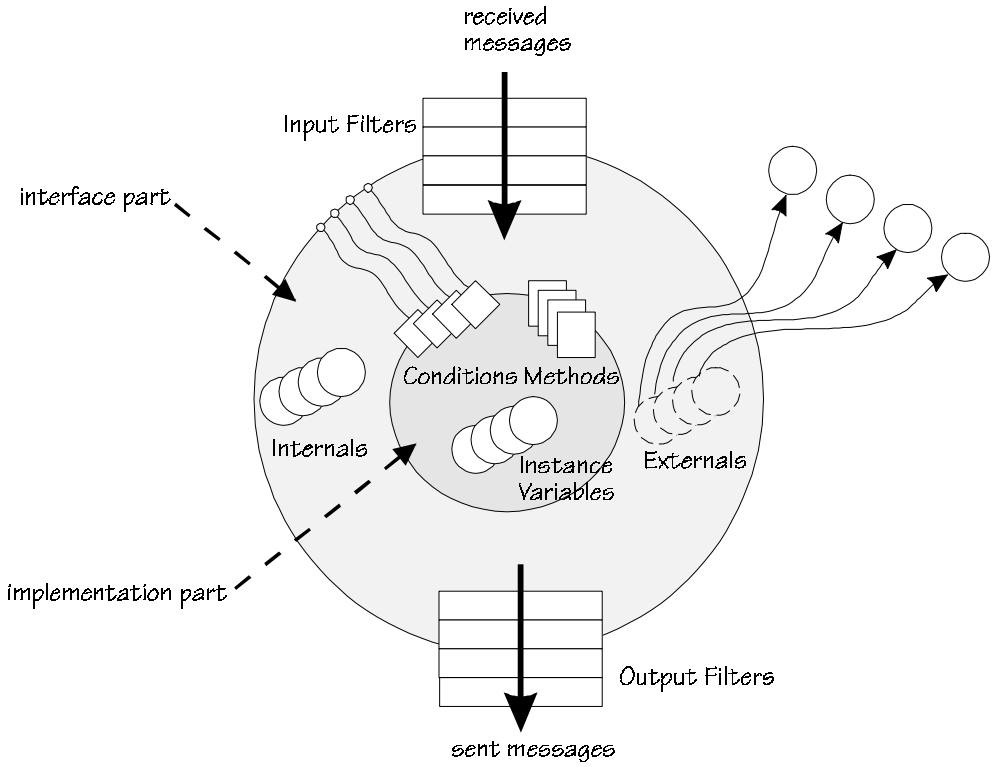


Figure 2.4.1 The components of the composition filters extension in the interface part.

The following components can be distinguished in the interface part of a composition filters object: *internals* and *externals*, which are nested objects, respectively references to external objects, *input filters* and *output filters*.

Further it can be observed in figure 2.4.1 that the conditions that are visible on the boundary of the implementation part (kernel object) are connected to the outside boundary of the interface part. This shows that the conditions are accessible on the interface of the object, albeit for a limited category of clients. The same is true for the input and output filters: these can be referred to by other objects as well.

Internals are fully encapsulated objects that are used to compose the behaviour of the composition filters object. The declaration of an internal object is analogous to the declaration of instance variables, and requires the specification of a label and an object class. Upon the creation of the object, its internal objects are automatically created as well.

Externals declare objects that exist outside of the composition filters object, but within its scope[1], such as for instance global objects. Within both the interface and the implementation part of the object all object identifiers must be locally resolvable. An identifier must either refer to a locally declared object or parameter, or to an object that is declared as an external. As a result, for all these objects the types are known at compile-time.

> The order in which nested objects are created and initial methods executed is as follows: (1) the externals of the object are located in the scope of the object, (2) the internals are created (because these are independent of the definition of the object), (3) the instance variables are created and (4) the initial method is executed. Note that this process is applied recursively; thus the initial methods of the internals and instance variables are executed before the initial method of the encapsulating object.

In the next subsection, the Sina syntax for specifying the interface part of an object is described, followed by a number of important applications of composition filters.

2.4.2 Defining the Interface Part in Sina

The declaration of the interface part conforms to the following template:

```
class ... interface
    comment ;
    internals
        ;
    externals
        ;
    conditions
        ;
    methods
        ;
    inputfilters
        ;
    outputfilters
        ;
end // class ... interface
```

All the components in a class definition, except for the begin and end clauses, are optional. In general, most classes will at least define methods, and one or more input filters. We will discuss each component, continuing our example of class Person from the previous sections.

The first clause defines the name of the class, which can be extended with a number of formal parameters between brackets, the *class parameters*. It is followed by the class comment, that describes the purpose and responsibilities of the class. For example:

```
class Person(pid:Integer) interface
    comment "This class defines the basic properties of person objects. It takes a single
             initialialisation parameter, the PID number";
```

The formal parameters are declared identically to other object and parameter declarations. A class parameter is similar to an internal declaration: both define a nested object. The only

[1] The exact scope rules are explained later in this chapter.

difference is the initialisation: the class parameters are initialised with values that are provided by the statement that causes the object creation[2].

Internals are object declarations, the creation of the encapsulating object will cause the creation of all the internal objects. The created internal object will be an instance of the declared class:

```
internals    // not in the Person example
    example1, example2  : Dummy;
    test : Person(5432866);
```

These internal declarations cause the creation of two internals of class Dummy, and another internal, test, is created as an instance of class Person. The latter example demonstrates how the values for class parameters are defined.

The declaration of externals takes the same form:

```
externals  // not in the Person example
    x, y : SomeSpecificClass;
```

This declaration does not cause the creation of a new instance of class SomeSpecificClass. Instead the declaration indicates that an object that is type-compatible with SomeSpecificClass is in the scope of the encapsulating object.

The conditions clause declares all the conditions that are used in the filters of the object, and makes them visible on the interface of the object. Conditions are either locally defined, in which case only the condition identifier suffices, or they are reused from other objects. In the latter case, the condition identifier is preceded by an object identifier. This must be either a class parameter, an internal or an external:

```
conditions
    isBirthday; isLocal;
    test.aCondition ;  // not in the Person example
```

The condition declarations are separated by semicolons. When reusing conditions from other objects the form "anObject.*" may be specified as well; it indicates that all the conditions on the interface of anObject are now available on the interface of the current object.

> Note that this does not break encapsulation because conditions are abstractions of the object implementation, and can be replaced by other condition implementations without consequences for the clients of the object. In addition, conditions are used only by 'reuse clients', such as subclasses.

The interface part of an object must also declare all the methods that are implemented by the implementation part and should become available on the interface of the object. They are declared as follows:

```
methods
    getFullName(Boolean) returns String;
    getPresent(Thing) returns Nil comment "accept the argument as a present";
```

---

[2] As the name class parameters suggests, they may also be used to specify parameterised, or generic classes (see for example [Meyer 86]). This is achieved by introducing dummy parameters of type Class, which can be used on the right hand side of object declarations within the object. Since we did not fully investigate the consequences of this technique, in particular with respect to type checking algorithms, we do not further consider this.

In the interface part only the header, or *signature*[3], of the methods is defined: the types of all the method arguments are specified, as is done for the return type. This makes the interface specification independent of a specific implementation part, although it causes some redundancy in the complete object specification. Optionally, a comment clause may be specified, which explains the responsibility of the method.

After the method declarations the input and output filters are declared. For example:

```
inputfilters
    select : Error = { True=>[*.getFullName]*.*, IsBirthday=>[*.getPresent]*.* };
    disp : Dispatch = { inner.* };
outputfilters
    continue : Send = { [x.*]y.*, * };  // not in the Person example
```

These input filters were discussed in the previous section, the output filter that is shown here gives an example of how output filters can be useful: a filter of type *Send* is declared, which is the equivalent of the *Dispatch* filter in the set of input filters: it cause the message to be transmitted to the target object.

The filter initialisation "{ [x.*]y.*, * }" consists of two filter element. The first element matches all messages that have a target equal to x, and then substitutes target y. Thus, all the messages that the object sends to the external object x, are redirected to external object y. All the other messages are transmitted unchanged, as they match the second filter element.

The complete interface definition of class Person is then as follows:

```
class Person(pid:Integer) interface
    comment "This class defines the basic properties of person objects. It takes a single
            initialialisation parameter, the personal ID number";
    conditions
        isBirthday; isLocal;
    methods
        getFullName(Boolean) returns String;
        getPresent(Thing) returns Nil comment "accept the argument as a present";
    inputfilters
        select : Error = { True=>[*.getFullName]*.*, IsBirthday=>[*.getPresent]*.* };
        disp : Dispatch = { inner.* };
end;  // class Person interface
```

We will refer to this class later in this section, as it will form the root in an inheritance hierarchy with several direct and indirect subclasses.

2.4.3 Data Abstraction Techniques with Composition Filters.

We will now discuss a number of data abstraction techniques that can be expressed with composition filters. The composition filters model does not provide techniques such as inheritance and delegation through dedicated language constructs, but uses the generic framework of composition filters to express these.

---

3 Not to be confused with the signature of an object, which is the collection of all the signatures of methods that are available on the interface of the object.

The data abstraction mechanism of the composition filters model is based on the principle of *composition*: rather than adopting a single data abstraction technique, the behaviour of an object is composed from the behaviours of one or more other objects. Object composition allows to express most well-known techniques, such as inheritance and delegation.

Because the *Dispatch* filter is the filter that eventually causes the dispatch of a message to the designated target, it is commonly used to express data abstraction techniques. This is also the case in this subsection. However, the essentials of the data abstraction techniques lie in the manipulation of the target and selector of messages, which can be done in most filter types.

The Example

To illustrate the various techniques we will further develop the Person class, taking this as the basis for modelling the people in a small business. Our example may sometimes look a bit contrived, as we combined a number of properties into a small set of classes, while keeping all classes very simple. In more complex applications, however, each of the data abstraction techniques that we describe may appear as a natural consequence of real-world modelling. This has been confirmed by pilot-studies such as [Greef 91], [Breunese 92], [Jonge 92] and [Tekinerdogan 94].

The following figure gives an overview of the objects and relations between them that we will discuss in this subsection:

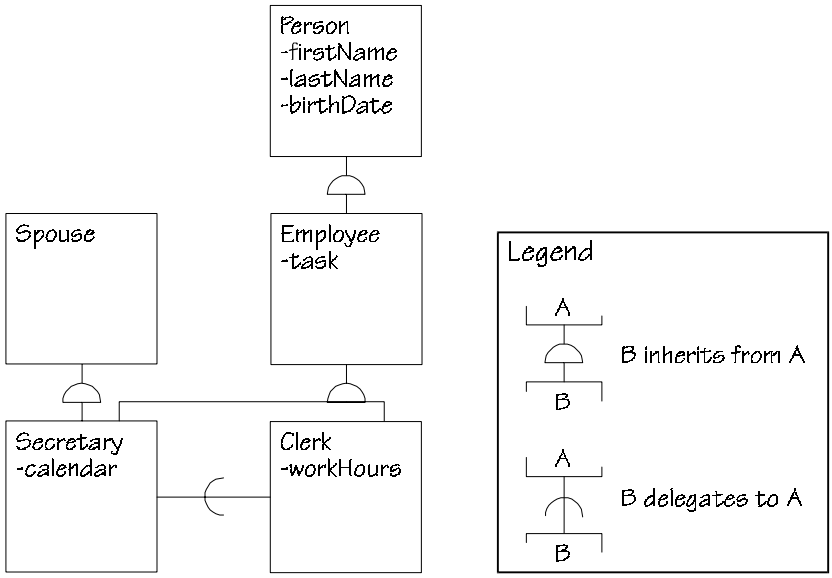


Figure 2.4.2 An overview of the reuse relations that are discussed in this subsection.

As an example we model the people working in a company. Class Person describes the basic aspects of people and has been described previously. To model the people that work in the company, a class Employee is introduced. This class inherits from Person as an employee has all the properties of a person and adds a few specific properties. In particular, each employee has a personal task. Thus, we require *single inheritance* to model this.

We distinguish between two types of employees: a Secretary and a Clerk. Both classes inherit from Employee, but Secretary in addition inherits from class Spouse. This requires

multiple inheritance. As a further refinement it can be noted that a person does not always behave like an employee. As a simplification we state that only from 9 to 5, the secretary behaves as an employee, and only during the remaining time, she can behave as a spouse. The technique to model this is called *dynamic inheritance*.

Furthermore, the clerks redirect all requests they receive for making appointments to the secretary, who keeps a central calendar. However, each clerk may have preferences, and has particular working hours. As will be demonstrated, this can only be modelled with the mechanism of true *delegation* [Lieberman 86].

Single Inheritance

The class Employee inherits from class Person, adds two new methods and one or more instance variables. The following interface definition realises this (the instance variables are defined in the implementation part, which we do not show here):

```
class Employee interface
    comment "this class, a subclass from class Person, models an employee. Each
             employee has its own task, which can be accessed with the methods
             <getTask> and <putTask>"
    internals
        pers : Person;
    methods
        getTask returns String comment "retrieves the string representing the current task";
        putTask(String) returns Nil comment "offers a string representing a new task";
    inputfilters
        inherit : Dispatch = { inner.*, pers.* };
end // class Employee interface
```

The most interesting parts of this code are the definition of the internals and the input filters; a single internal is declared, that is labelled pers and is an instance of class Person. The class defines a single filter, of type *Dispatch*. The first filter element, "inner.*" allows the execution of all local methods, getTask and putTask in this particular case.

The second filter element, "pers.*" substitutes the target pers for all messages with a selector that is in the signature of pers. Because pers is declared as an internal of class Person, this happens if the message selector is defined by class Person. As a result, the subsequent dispatch of the message will cause it to be transmitted to[4], and executed by, the internal pers.

The mechanism that we just described simulates the conventional inheritance mechanism. We will show this by considering the following three properties of inheritance: data structure inheritance, method inheritance and dynamic binding.

A brief discussion of these three properties: Data structure inheritance means that the data structure, as it is formed by the instance variables, that is defined by a superclass P is replicated for all instances of its subclass B. Note that strong encapsulation may prohibit direct access to the inherited data structure (see e.g. [Snyder 86], [Micallef 88]).

4 Such a message dispatch has in fact the same semantics as delegation [Lieberman 86].

Method inheritance means that all the methods that are accepted by instances of the superclass, are also available on the interface of the instances of the subclass, even though the methods are defined in the superclass only. However, it is possible to redefine (or override) the definition of an inherited method with a new definition in the subclass.

Dynamic binding is an essential property of the inheritance mechanism. Due to dynamic binding, within the superclass definition one can refer to a method that is defined by a subclass. This requires a form of self-reference that can dynamically change at run-time; depending on the class of the object that received an -inherited- message, calling a method may require differing implementations.

Data structure inheritance is achieved through the internal declaration: this causes each instance of Employee to contain an instance of class Person. As a result, the data structure defined for persons is replicated for all employee objects. Because the data structure of class Person is encapsulated within the pers object, it cannot be accessed directly (i.e. strong encapsulation).

Method inheritance is achieved through redirection of the message in the dispatch filter. As a result, all the messages that are implemented by the Person class, are executed by the pers internal object. However, all that the client objects can observe is that the employee object supports all the methods that are defined by class Person, and adds a few more.

Note that the subclass can easily override methods defined by class Person; because of the ordering of the filter elements in the dispatch filter, it is first tried to match an incoming message with inner. Only when that fails, it is tried to match with the second filter element, that represents all the methods defined by the superclass.

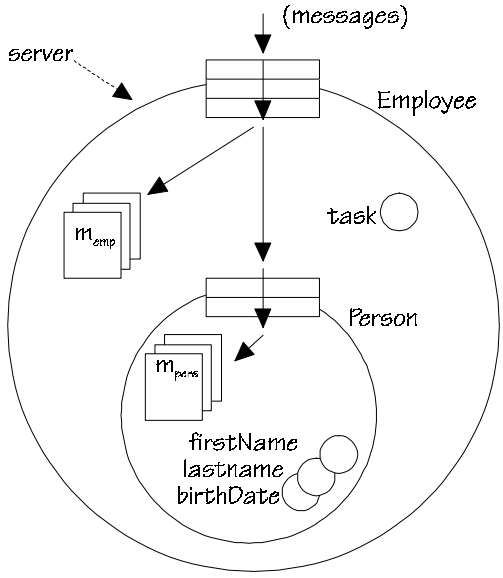


Figure 2.4.3 Inheritance in the composition filters model.

Dynamic binding is based on dynamic self-reference, which means that it is possible to refer to the original receiver of the message. In Sina, this is achieved through the pseudo-variable *server*[5]. This pseudo-variable always refers to the object that first received the message. By

5 This pseudo-variable is comparable to *self* in Smalltalk, *this* in C++ and *current* in Eiffel.

sending a message to *server*, an inherited method in the superclass can invoke a method defined in one of its subclasses.

The specification of inheritance is illustrated by figure 2.4.3. This figure shows an instance of class Employee that encapsulates an instance of class Person as an internal. The data structure (i.e. instance variables) defined in class Person, as well as the additional data structure defined by class Employee, are both encapsulated within the instance of class Employee. The methods defined by class Person, mpers, are available on the interface of the employee object. An incoming message that is defined by the superclass will be redirected to the person object, which triggers the execution of the corresponding local method. The pseudo-variable *server* will refer to the employee object, as this is the receiver of the message. The observed behaviour of the employee object is that it supports both the messages from class Person, and a number of own, employee-specific methods. This technique for simulating inheritance is also called delegation-based inheritance.

Multiple Inheritance

We demonstrate multiple inheritance with the following definition of a secretary object:

```
class Secretary interface
    comment "This class models a secretary; it inherits from class Employee and class
             Spouse, and maintains a shared calendar through the schedule method";
    internals
        work : Employee;
        private : Spouse;
    methods
        schedule(Appointment) returns Boolean;
    inputfilters
        multInh : Dispatch = { inner.*, private.*, work.* };
end // class Secretary interface
```

Class Secretary defines two internals, one for each superclass: internal work is an instance of class Employee, and the internal private is an instance of class Spouse. We show only a single method, named schedule, that takes an appointment as an argument, and returns a Boolean indicating successful scheduling. The class defines a single input filter, of type Dispatch, with the initialisation "{ inner.*, private.*, work.* }". Incoming messages are first tried to match with the signature of inner[6], then with the signature of private, and finally with that of work.

The effect of this filter expression is that the object can accept three kinds of messages: firstly, all the messages that are locally defined (in this case, only schedule). Then all the messages defined by class Spouse, and finally the messages that are provided by class Employee. This mechanism simulates multiple inheritance, as can be motivated by adopting the same arguments we used for explaining single inheritance.

---

6 It may seem a bit overdone to use the '*' wild card for a single method, but this is more a matter of style; it expresses that all the locally defined methods are made available on the interface of the object. Further, this immediately takes care of methods that are added in the future.

Multiple inheritance may cause conflicts when multiple superclasses offer methods with the same name. Various solutions for this problem have been proposed, for example method renaming in [Meyer 88] and linearisation of the inheritance graph in CommonLoops [Kempf 87]. The composition filters model avoids inconsistencies through the filter matching rules: a received message is accepted at the first filter element that matches. In our example, methods provided by Secretary have precedence over methods of Spouse, which in turn precede over the methods of Employee.

To simulate inheritance, filters specify the assignment of a new target to a received message, based on the selector of the message. Filters allow for specifying this on the level of individual messages, if necessary.

As an alternative, methods with conflicting names can be renamed, to allow both inherited methods to be accessible. The following variation to the multInh filter of the Secretary class shows both renaming of conflicting methods and the ordering of filter elements to select specific methods from one of the superclasses:

```
multInh' : Dispatch = { inner.*, [callPrivate]private.call, [call]work.call, private.*, work.* };
```

We assume, for the sake of this example, that both class Spouse and class Employee provide a call method. The second filter element renames message callPrivate to call, and substitutes private as its target. The third filter element ensures that the call message is inherited from class Employee. Note that this filter element is necessary, because otherwise the call method from the Spouse class would be selected, in the fourth filter element.

It should be noted that the mechanism for multiple inheritance may bring problems in the occurrence of shared ancestor classes. The problems are due to the replication of the data structure of the parents. Consider for instance that in our example, the class Spouse inherits from class Person as well. In this case a single instance of Secretary contains two instances of class Person: one as an internal of work, and one as an internal of private. This is shown in the following figure:

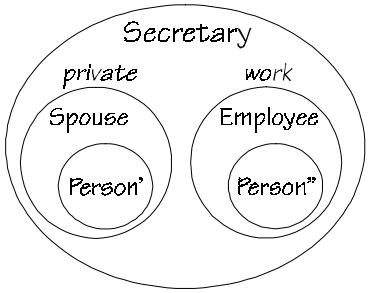


Figure 2.4.4 The object nesting structure of multiple inheritance with shared parents.

If a message is received that is inherited, it is dispatched to either the Spouse or the Employee, and will thus update and access either Person' or Person''. This can cause two problems: when some of the inherited messages are executed by Person' and some by Person'', this may lead to inconsistencies. On the other hand, when both the methods of Spouse and Employee refer to the state of their respective superclass through message sending, in at least one case this state will not be up-to-date.

This problem can be partially solved by a mechanism that is similar to *virtual superclasses* in C++ (see [Ellis 90] for a discussion of the problems of virtual classes). This mechanism allows for specifying a superclass to be virtual, which causes repeated inheritance from that class to

result in only a single instantiation of the data structure defined by the class. In our example this would cause Person' and Person'' to be references to a single instance of class Person.

However, this is not a truly satisfactory solution, as the decision for making a superclass virtual is made at a place where the problem is not relevant yet: only in the class that defines multiple inheritance, such as Secretary in our example, this problem appears. It would make sense to take a decision about how to handle repeated ancestors only at this place. However, encapsulation should not be violated by referring to the inheritance hierarchy other than the direct superclasses. The composition filters model therefore does not provide such a mechanism, but leaves it to the programmer to avoid inconsistencies, for example by restructuring the system so that Spouse and Employee share a single instance of class Person.

Dynamic Inheritance

The previous example showed that we can model a situation where an object has more than one role. This is a common modelling issue, but often the roles of an object depend on the status of the object. This can be the context of the object or the internal state of an object. In particular for objects that travel through various parts of a system, changing roles frequently occur. In subsection 2.4.4 it is discussed how parts of the behaviour of an object can be temporarily hidden. Here we discuss the problem of dynamically changing inheritance relations.

We adopt the term *dynamic inheritance*[7] to designate a mechanism that allows for enabling and disabling an inheritance relation with a superclass at runtime. The starting point is the definition of a fixed set of superclasses, by defining the corresponding internals. In the filter specification each inheritance relation is then constrained by adding a condition to the filter element.

We demonstrate this by changing the secretary example, so that a secretary object inherits only from class Employee during business hours, and inherits only from Spouse after business hours:

```
class Secretary2 interface
    comment "this class models a secretary that dynamically inherits from class Employee
                and class Spouse, depending on the time of day";
    conditions
        WorkHours; OffTime;
    internals
        work : Employee;
        private : Spouse;
    methods
        schedule(Appointment) returns Boolean;
    inputfilters
        dynInh : Dispatch = {WorkHours=>inner.*, OffTime=>private.*, WorkHours=>work.*};
end // class Secretary2 interface
```

The mechanism for dynamically changing the inheritance relations is achieved by constraining the filter expressions with conditions. Two conditions, respectively WorkHours and OffTime are defined by the object (and implemented in the implementation part, which is not shown here). Although the two conditions are mutually exclusive in this example, this

7 The language SELF [Ungar 87] supports dynamic inheritance as well.

is not necessary in the general case; the inheritance relations may dynamically vary between no inheritance relation at all, to a situation where all possible inheritance relations are enabled.

For consistency, the methods that are defined locally, which also deal with the working role of the secretary object, are constrained similarly to the inherited methods in the first filter element. This is not dynamic inheritance, though. To keep the filter specification small, we ignored the renaming of the `call` method here.

> Dynamic inheritance does not very well combine with type-checking in the sense that the type-checking mechanism can no longer ensure at compile time that all the message invocations in a type-checked program will be acceptable by the receiver object[8]. The reason for this is that the condition expressions can be arbitrarily complex, and thus in the general case it cannot be predicted whether an inherited method is available when the message is sent. Type-checking is based on the largest possible signature, i.e. when all inheritance relations are enabled. We do not consider this as a severe restriction, as we assume that an implementation without the dynamic inheritance mechanism will behave similarly, but implement explicit tests in the bodies of methods.

An important application for dynamic inheritance is to realise *alternative implementations*. Alternative implementations means that the same methods, received by the same object, may have different implementations. This mechanism can be very advantageous to achieve truly reusable frameworks (cf. *black-box frameworks* in [Johnson 88a]). When the interface of the object does not change over the life-time of the object, dynamic inheritance can be simulated within the conventional object model. But the conventional object-oriented model does not support changing interfaces of objects[9].

Delegation

The composition-filters model supports both inheritance and delegation. Delegation is particularly useful to express behaviour sharing in combination with data sharing: inheriting from a class replicates the data structures, providing each instance of the subclass with its own copy of the data. To share data, objects may refer to global objects that are accessible by more than one object. Such shared objects must be addressed through message invocations.

Message sending is different from behaviour reuse, though. This is described by the so-called *self* problem [Lieberman 86]. When reusing a service, the server object must be 'part of the extended identity' of the client object. In [Aksit 92b] it is shown why both inheritance and delegation are needed[10].

---

8 The work on type inferencing in [Agesen 93] does address the issue of dynamic inheritance in the dynamically typed language SELF, we have not fully investigated the applicability of this work to Sina, however.

9 In object models that provide delegation, dynamic inheritance *can* be simulated, though.

10 In a strict sense, delegation only is sufficient, as delegation can be used to simulate inheritance. In fact, the message dispatch implemented by the *Dispatch* filters has semantics that are similar to delegation. We support inheritance because it is an important modelling tool.

It is now shown how delegation can be expressed in the composition filters model. We give the definition of a Clerk object, which is a subclass from class Employee. All clerk objects delegate the schedule message to a shared secretary object, as follows:

```
class Clerk interface
    comment "this is a clerk object, a subclass of Employee. The schedule message is
             delegated to an external secretary object.";
    internals
        empl : Employee;
    externals
        administrator : Secretary;
    methods
        getWorkingHours returns WorkingHours
            comment "this returns the regular working hours of the clerk";
        setWorkingHours(WorkingHours) returns Nil
            comment "set new values for the working hours of the client";
    inputfilters
        delegate : Dispatch = { inner.*, empl.*, administrator.schedule };
end // class Clerk interface
```

Class Clerk defines an internal, which is an instance of its superclass, Employee, and two methods; getworkingHours and setWorkingHours. The class also defines an external, named administrator, that is declared to be of type Secretary. The dispatch filter of class Clerk consists of three filter elements. The first filter element makes all the locally defined methods available on the interface, the second element realises inheritance from class Employee.

The last filter element, "administrator.schedule" specifies that for all messages with selector schedule the target will be substituted with administrator, provided that schedule is in the signature of administrator. This is obviously the case, since class Secretary offers the schedule method on its interface. As a result of this filter element, received schedule messages are redirected to the administrator object.

Thus, the administrator object takes care of scheduling appointments for instances of class Clerk. The scheduling of these appointments is partly determined by the calendar that is maintained by the administrator object. Multiple instances of class Clerk that operate in the same context, for example within the same encapsulating company object, will all refer to the same administrator object. Thus, they all share the same calendar data.

Scheduling of appointments by the secretary takes into account the working hours of the clerk that originally received the schedule message. The secretary object can refer to this through the pseudo-variable server, which always refers to the object that first received the message leading to the current execution. A dispatch in the filters of an object does not affect the value of server, only a message invocation changes it.

To conclude, the presented example implements the delegation mechanism, and thus realises sharing of both the behaviour of the secretary object administrator and its internal state, in particular the calendar. The self problem is solved through the server pseudo-variable. The same variations as for inheritance can be applied to delegation. Multiple delegations are possible, and dynamically changing delegation relations are possible as well.

In this subsection we have demonstrated a number of data abstraction techniques, all based on the application of the dispatch filter. These techniques are partly based on the functionality of the dispatch filter, and partly on the matching and manipulation of the target and selector of messages. The latter is allowed in filters of other types as well. In the case that a message does not match any of the filter elements of a dispatch filter, an error will occur. Therefore a filter of type Dispatch will always be the last filter in the set of input filters[11].

2.4.4 Multiple Views & Preconditions with the Error Filter

Another filter type is introduced now, and a few possible applications are described. The filter type we are concerned with is type Error. The semantics of the error filter are very simple: when a message is accepted (i.e. it matches for one of the filter elements), it can simply proceed to the next filter, while the substitutions that are made to the message are retained. When the filter rejects the message, an error will occur, and the execution is halted.

The most straightforward application of the error filter is to screen incoming messages, and enable only a restricted set of messages to proceed to subsequent filters. This was already demonstrated in the previous section. As an example, class Clerk is extended with an error filter:

```
class Clerk interface
    comment "This is a clerk object, a subclass of Employee. The messages putTask and
            getPresent are excluded from the interface of the object";
    ... // some parts that were defined in the previous subsection are skipped here
    inputfilters
        exclude : Error = { ~>{putTask, getPresent} };
        delegate : Dispatch = { inner.*, empl.*, administrator.schedule };
end // class Clerk interface
```

The error filter, named exclude, consists of a single filter element, which starts with the exclusion operator. This means that the filter element matches for all messages except those defined between the curly brackets: putTask and getPresent. Thus, when one of these two messages is received, the filter will generate an error. As a result, both these methods, which are otherwise inherited from class Employee and Person, are not available for the clerk objects now. This demonstrates that we can remove methods that are defined by one of the superclasses from the interface of the object.

Multiple Views

We noted before, in the discussion on dynamic inheritance, that an object can take several roles during its life-time. These roles can be determined by the internal state of the object itself, or may depend on the environment. In particular, an object is likely to behave differently for different clients (see also [Pernici 90], [Hailpern 90], [Aksit 92a] and

[11] We have also been experimenting with dispatch filters that pass the message when rejected, thereby allowing for multiple dispatch filters in a single set. The dispatch semantics adopted in this thesis give more safe and well-structured filter specifications, though.

[Aksit 92b] for discussions on this topic). Some messages should be invoked only by certain clients, and are not acceptable from all clients.

The Clerk class is modified to demonstrate this: assume that the putTask message should only be sent by an object that is a Manager, and that the message getWorkingHours is only acceptable from a Manager or the administrator object. These situations can be checked by looking at the pseudo-variable sender. This pseudo-variable reveals the identity of the object that sent the message, based on the value of the server pseudo-variable just before the message invocation.

The value of sender can be tested in the definition of a condition in the implementation part of an object:

```
conditions
    SentByManager begin return sender.isSubTypeOf(Manager); end;
    SentBySecretary begin return sender=administrator; end;
```

The isSubTypeOf message, that is available for all objects, returns a Boolean value indicating whether the receiver is a subtype of the class provided as an argument. This has the advantage that specialisations of the Manager class are allowed as well.

We can then realise our constraints on received messages with the following filter definitions:

```
inputfilters
    select : Error = { SentByManager=>putTask,
                       {SentByManager, SentBySecretary}=>getWorkingHours,
                       True~>{putTask, getWorkingHours}  };
    delegate : Dispatch = { inner.*, empl.*, administrator.schedule }; // unchanged
```

The select filter consists of three elements: the first filter element defines that the putTask message is only accepted when the SentByManager condition is satisfied. The second filter element states that the getWorkingHours message is only accepted when either the SentByManager or the SentBySecretary condition is satisfied. The last filter element enables all messages, except putTask and getWorkingHours to be accepted under all circumstances.

This example shows how the interface of an object may vary depending on the identity of the client that sends the message. This implements different views of the -interface of the- object for different clients.

Preconditions

The selective admission of messages can be based on other criteria than the identity of the sender of the message: by providing another implementation of the conditions, a different selection criterion can be effectuated. This mechanism can be used to implement preconditions for methods, similar to assertions in Eiffel[12].

[12] However, we cannot easily implement post-conditions and class invariants, and lack the language support for expressing assertions that Eiffel provides. For details see e.g. [Meyer 88] or [Meyer 92].

Consider for example the Employee class that was defined in the previous subsection. This class provides two methods, respectively getTask and putTask. Preconditions can now be defined for each of these messages, in order to avoid inconsistencies: getTask should not be accepted when the task instance variable is an empty string, and putTask should not be allowed when the task instance variable is *not* an empty string. The following updated class definition realises these preconditions:

```
class Employee2 interface
    comment "This class models an employee. Each employee has its own task, which can
             be accessed with the methods <getTask> and <putTask>. "
    conditions
        NoTask;  // true when no task to do is currently defined.
        TaskLeft; // true when there are one or more tasks left to do.
    internals
        pers : Person;
    methods
        getTask returns String comment "retrieves the string representing the current task";
        putTask(String) returns Nil comment "offers a string representing a new task";
    inputfilters
        preConditions:Error={TaskLeft=>getTask, NoTask=>putTask, ~>{getTask, putTask} };
        inherit : Dispatch = { inner.*, pers.* };
end // class Employee2 interface
```

The important part of this definition is the error filter preConditions; it associates condition TaskLeft as the precondition of the getTask message, the condition NoTask is the precondition of the putTask message, all other messages can always pass the error filter.

2.4.5 Abstracting Object Interactions with Meta Filters

The concept of *Abstract Communication Types* (ACTs) was introduced in [Aksit 89], and presented in the form of composition filters in [Aksit 92c] and [Aksit 94a]. An ACT implements coordinated behaviour between objects; it is a first-class object representation of the message interaction patterns. As a result, it can express patterns of communication or message interaction protocols.

An ACT class is an ordinary Sina class with the same syntax and semantics. What makes a class an ACT class is the way its behaviour is composed with its participating objects: an ACT manipulates first-class representations of messages. The participating objects are defined such that their message interactions are intercepted and transformed into first-class representations. This is achieved through the introduction of a new filter class, the *Meta* filter.

A *Meta* filter has a structure similar to the *Dispatch* filter. If the received message is not accepted by the meta filter it is passed to the next filter. However, if the received message is accepted by the meta filter the message is first converted to an instance of class *Message* and then passed as an argument of a new message to the ACT object. This conversion operation is also known as *reification*. The ACT object can retrieve the relevant information about the message from the argument. An ACT can also modify the contents of the message by invoking the operations of class *Message*. Finally, an ACT can convert an instance of *Message* back to a message execution.

As an example, we create a subclass of Clerk so that all the messages sent and received by clerk objects are logged by a global ACT object. The LoggedClerk class requires the addition of a single input filter and an output filter, and an external that declares the ACT object:

```
class LoggedClerk interface
    comment "This is a subclass of clerk of which all the incoming and outgoing messages
            are logged by the external ACT object named bigBrother";
    internals
        clerk : Clerk;   // inherit from clerk
    externals
        bigBrother : LogACT;   // this declares an external ACT object
    inputfilters
        reifyIn : Meta = { [*.*]bigBrother.logMessage };  // reify and send message to the ACT
        inherit : Dispatch = { clerk.* };
    outputfilters
        reifyOut : Meta = { [*.*]bigBrother.logMessage }; // reify and send message to the ACT
end // class LoggedClerk interface
```

All the received messages must first pass the reifyIn filter. This meta filter matches with all messages, due to the matching part "[*.*]". Thus all messages will be reified, and supplied as the argument of the message logMessage that is sent to the external object bigBrother. We call the message logMessage a *meta-message*, as it contains meta-information about the original message. The ACT object bigBrother logs the (meta-) message and takes care that the active message resumes its execution, starting with the subsequent filter. All the outgoing messages pass the outputfilters of the object, and are at that point intercepted by the reifyOut filter. This behaves exactly the same as the reifyIn filter, and sends all the meta messages to the bigBrother ACT. This is an instance of class LogACT, that is defined as follows:

```
class LogACT interface
    comment "this class logs all the received meta-messages, and fires them again" ;
    methods
        logMessage(Message) returns Nil;
    inputfilters
        disp : Dispatch = {inner.* };
end; // class LogACT interface

class LogACT implementation
    comment "we implement logging by adding the first class message representation at
                the end of the <log> collection object. Then the message is fired, causing it to
                resume its execution:"
    instvars
        log : OrderedCollection;
    methods
        logMessage(mess : Message) returns Nil
            begin
                log.addLast(mess);
                mess.fire;
            end;
end; // class LogACT implementation
```

A typical property of an ACT is that it can handle meta-messages, i.e. messages with an argument of class Message. In this example this is the logMessage method. To keep the example simple, logging of messages is implemented by adding the first-class message representations to an ordered collection. The second statement of the logmessage method invokes an operation on the first-class message, causing it to resume execution: the fire operation re-creates a real, active message from the current values of the meta-message, and fires the newly created message, so that it can resume its execution (perhaps at another target, or with a different selector).

The interface of class Message offers a number of operations. Firstly, there are operations to get and set the values for the sender, the target, self, the message selector and the message arguments. Secondly, there are a few operations to manipulate messages: the fire method, which creates a message execution from a meta-level representation of messages. The copy method returns a copy of the message, but without its calling context[13]. The method reply takes a single argument and sends this argument as a reply to the sender that is defined in the first-class message. The interface of class Message is given in the appendix.

> The currently defined operations on first class message representations may lead to problems when not applied carefully. For instance, the identity of the sender of a message must be maintained when the message is intercepted by an ACT and then fired again. These problems can be avoid through a proper redesign of the operations on first class messages, although this requires some limitations on the manipulation of messages. Such a redesign is shown in [Wakita 93]. We leave the responsibility to the programmer.

Abstract communication types offer a number of advantages:

- *Expressive power*: ACTs increase the expressive power of the model, as they allow for tailoring the semantics of message passing.
- *Complexity:* ACTs can make the complexity of programs manageable by moving the interaction code to separate modules. This allows for reducing the number of inter-module relations and hiding communication details. ACTs can be applied to construct layered systems.
- *Separating functional- and interaction-code:* ACTs promote the migration of interaction code to separate modules, thereby improving maintainability and reusability.
- *Reusability:* Different types of classes may have common patterns of communication and synchronisation. Such commonalities can be abstracted using ACTs. Programmers may apply object-oriented techniques, such as inheritance and delegation, to achieve a more systematic reuse of ACTs, and thus of interaction code.
- *Enforcing invariant behaviour:* It is easier to enforce the invariant behaviour among objects if there is a module explicitly representing this behaviour.

However, ACTs should be applied with care: ACTs are only appropriate as a modelling tool, and should thus only be applied for modelling real-world or design entities. Carelessly

---

[13] This is to avoid that multiple replies are created for a single message invocation.

shifting interaction code from participating objects to ACTs may violate the important object-oriented principle that every object is self-sufficient and independent. This would result in bad design characteristics. Nevertheless, if applied with care, ACTs offer a powerful technique for addressing certain types of modelling problems [Aksit 92b].

2.5 Further Language Aspects

This section briefly describes a number of language aspects that have not been covered yet in the previous sections. This includes type-checking, scope rules, an overview of the pseudo-variables, and the mechanism of *atomic delegations*. These aspects complement the description of the language that has been given so far.

Type-Checking

The general aim of type-checking, phrased in object-oriented concepts, is to ensure that whenever a message is sent to an object, this has indeed the desired effect. It is very well possible that different objects (i.e. instances for different classes) can all handle a message in a satisfactory way. This is even essential, as it allows for replacing an object in an application with a specialisation of it, without modifying other parts (object definitions) of the program. Thus the most important goal in type-checking is to guarantee substitutability, or *conformance* of objects.

The preferred approach to type-checking is to determine semantic substitutability of objects, in the sense that an object *S* may replace an object *T* (i.e. *S* is a subtype of *T*) only if the effects of the messages that are sent to *S* are semantically compatible with the effects of sending the same messages to *T*. As noted in [Micaleff 88], this is only possible when based on precise formal specifications of the semantics of an object. However, as this approach is not feasible, practical type checking algorithms are based on the conformance of the interfaces of objects[1]. Type checking is then reduced to the following goal: to ensure that every message that is sent to an object, will indeed be supported by the object.

Sina is a *strongly-typed* language: assignments, objects that are passed as parameters and return values must satisfy their respective typing constraints. For each instance variable, temporary object, message argument and message return value its type must be declared. It is attempted to detect possible violations of the typing rules as much as possible at compile-time. This is also referred to as *static typing*. However, in the general case, it is not possible to fully type-check a program at compile-time. Thus, a part of the type checking can only be performed at run-time (called *dynamic typing*).

In general static type-checking is considered essential for the construction of reliable software; in dynamically typed software that has not been not exhaustively tested, type conflicts may be encountered at run-time. On the other hand, dynamically-typed systems are considered to be more suitable for (rapid) prototyping of software.

The type-checking rules in Sina are based on the rule of *contra-variance*, and ensure that a receiver in a message expression will always support the message that is sent. For instance Emerald [Black 87], provides the same rules. The rules are defined in terms of the *signature* of objects. The signature of an object is the set of messages that it supports, each with the types of their arguments and return type.

---

1 The approach taken in [America 90] is to characterise the semantics of the object by user-specified identifiers.

Definition 2.5.1: Type compatibility rules

An object *S* is a subtype of an object *T* if, and only if, it satisfies all of the following rules:

(i) *S* supports at least all the messages that *T* does.

(ii) For each message of *T*, the corresponding message of *S* (with the same selector) has the same number of arguments.

(iii) The types of the arguments of a message *m* of *T* conform to the types of the arguments of *m* defined by *S*.

(iv) The type of the result of a message *m* of *S* conforms to the type of the result of m defined by *T*.

With these rules, subtype compatibility is independent of the inheritance hierarchy. This is desirable [America 90], but not the case for the majority of the object-oriented languages (e.g. Eiffel, C++). To relax the typing constraints, the special type Any is available: *any* type is a subtype of Any. This can be very useful for instance in the prototyping phase of software construction. In some cases, the type of an object is not relevant, for example in objects that define storage structures, the type of the objects that are stored is not important[2].

> Recent advances in type inferencing, as documented e.g. in [Palsberg 91], [Agesen 93], may bring type inferencing closer to practical application. An important problem with these approaches, however, is that they work only on a closed set of application classes. This is both a problem because of the computational effort, and because it clashes with modular development and incremental compilation of software.

The signature of a Sina object is formed by combining the set of messages defined by an object with the messages that the object inherits or delegates. The latter is derived from the dispatch filter of an object, by stripping all its conditions (i.e. assuming these are true). Since internals and externals are typed, their signature can be determined as well.

At instance creation time, the signature of the object is constructed and the types of externals are verified. The signature of an object is also used for matching and substitution in filters. This late construction of the final signature supports open-endedness, as the evolving interfaces of e.g. superclasses and delegated objects are incorporated into the interface of the object.

Scope Rules

The scope rules define how object names are bound to objects. The concept behind the scope rules is based on the nesting of objects and adheres to the rule of encapsulation. This means that object boundaries are transparent when looking from the inside to the outer context, but that it is impossible to see what is within an object boundary from the outside of that object. Figure 2.5.1 illustrates this concept.

---

[2] The use of genericity is not always satisfactory in such situations, because of the following problem: consider a collection object with a get and a put operation that stores objects of a type *T* -indicated as coll(T)- that is defined through parameterisation. The problem is that, for different types *S* and *T*, coll(T) can never be a subtype of coll(S), even if *T* is a subtype of *S*. This is due to the contravariance between arguments (e.g. of put) and return types (e.g. of get).

The figure shows an object, aSecretary, with an internal employee. From the point-of-view of a method of the employee object, the following objects are visible: the instance variables of employee, the internals of employee, the internals of aSecretary (including employee), and the objects in the context of aSecretary (including aSecretary). The instance variables of aSecretary, within its implementation part (not shown in the figure), are hidden by the encapsulation boundary of the implementation part, or kernel object.

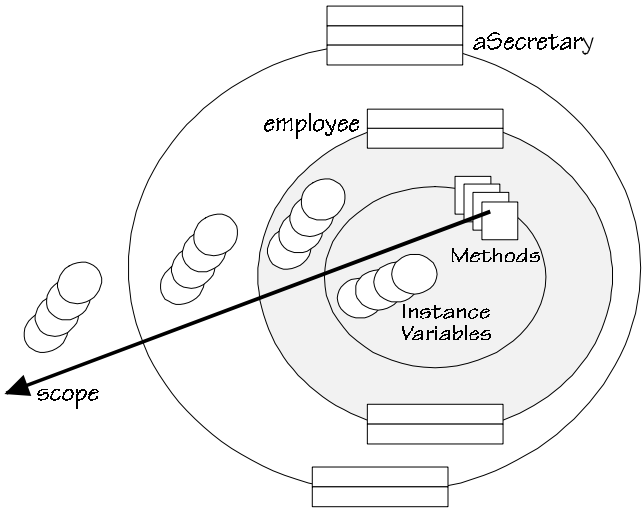


Figure 2.5.1 A schematic representation of the scope rules.

An important property of these scope rules is that they do not require unique names for all objects. For example, consider an application class Clerk that delegates to an external object administrator. In principle instances of Clerk share administrator with all other instances of the class. However, this is only true if all instances of the class have the same external object in their individual scope. An object nesting structure as given in the following figure shows that this is not always the case:

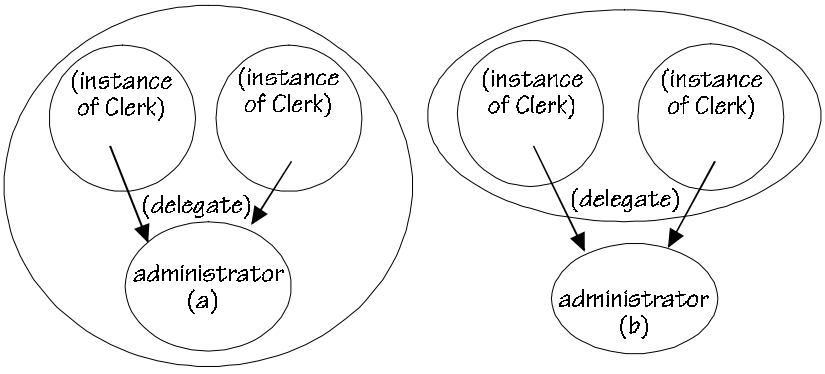


Figure 2.5.2 An object configuration demonstrating multiple objects with the same name.

In this figure two objects that are labelled administrator are shown, but each in a different context. Note that in the absence of the administrator object (a), the administrator (b) would be used by all four instances of Clerk in the figure. But in the absence of administrator (b) the two right instances of class Clerk in the figure would find no administrator in their scope.

Pseudo-Variables

In the previous discussions a number of pseudo-variables were discussed. Here we summarise their definitions, based on the following example of an execution thread:

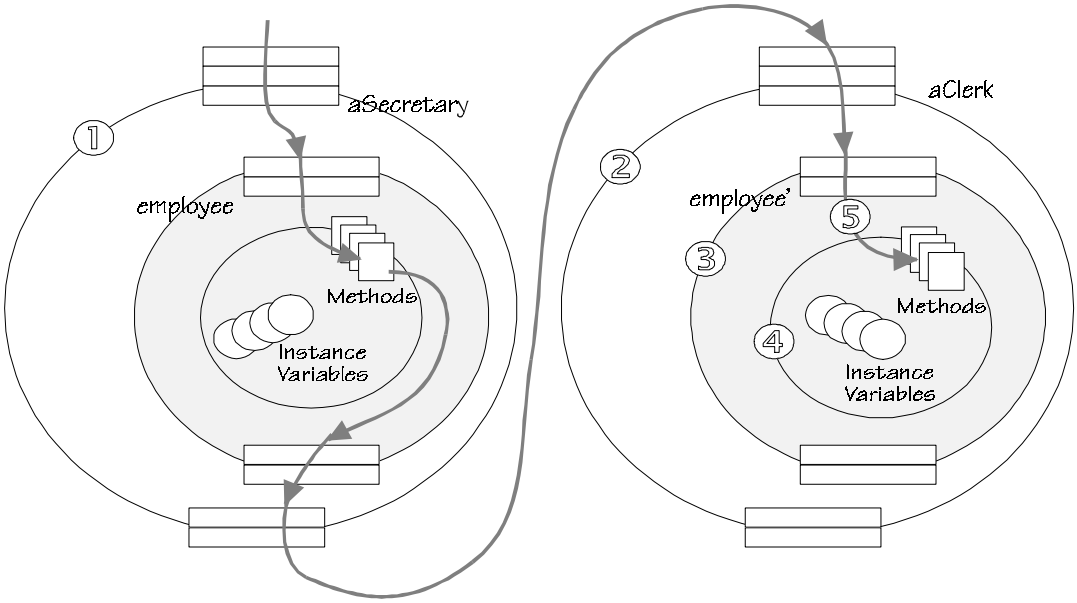


Figure 2.5.3 An illustration of an execution thread with the current pseudo-variables: ①=sender, ②=server, ③=self, ④=inner, ⑤=message

The figure shows an execution thread of a message that is sent to object aSecretary, which dispatches the message to its internal employee, where it triggers the execution of a method. During the execution of this method, a new message is sent to the object aClerk, which dispatches the message to its internal employee' at the input filters. This causes the execution of a local method of the employee' object. The numbers ① to ⑤ at that moment correspond to the following pseudo-variables:

① *sender*: This is the object that was responsible for sending the currently executed message (i.e. the message referred to by pseudo-variable *message*). This corresponds to the value of *server* at the moment of the message invocation. In the example, the sender pseudo-variable does not refer to the employee object, although the message invocation was executed during the execution of a method of employee, but it refers to the aSecretary object instead. Obviously, it is in general only known at run-time what the sender object, or even the type of the sender is (as an object can have several clients).

② *server*: The original receiver of the message that caused the current execution is referred to by *server*. A message invocation modifies the value of server, but a dispatch at the filters does not modify it. Thus, in our example, the message that is sent to the aClerk object is first dispatched to the employee' object, which triggers the execution of one of its methods. The *server* pseudo-variable thus refers to aClerk, and this is not affected by the subsequent dispatches at the input filters. The value of *server* can only be safely determined at run-time, due to dynamic binding. *Server* corresponds to *self* in Smalltalk, *current* in Eiffel and *this* in combination with virtual functions in C++. The usage of *server* is essential for extensibility, as it allows for overriding methods in (future) subclasses.

③ *self*: This refers to the object that executes the current method. A message that is sent to *self* must pass the output filters and then the input-filters of the object. Thus, the messages that are inherited or delegated by the object can be called through *self*. *Self* is statically bound, it supports efficiency and safety, because a message sent to *self* can be resolved at compile-time. This allows for the generation of more efficient code, because the receiver object and its type can be determined at compile-time. It is safer because the designer of the class has full control with respect to which method is executed as a result[3], and this cannot be overridden in a future subclass. In the example, the method that is being executed is defined by employee', thus *self* refers to employee'.

④ *inner*: The kernel, or implementation part of the current object can be referred to with the *inner* pseudo-variable. A message to *inner* will not pass any filters, and must conform to a method that is defined in the implementation part of the current object. Some of these methods may not be declared in the interface part at all, in which they case they are only accessible by the object itself through *inner*. This is similar to the definition of 'private' member functions in C++.

⑤ *message*: This pseudo-variable refers to the message that caused the current execution. The *message* pseudo-variable is an instance of class *ActiveMessage*, which allows for retrieving the properties of messages, but no modifications. Other properties of the message that can be accessed are the selector ("message.selector"), and the arguments (through an index: "message.argAt(..)").

In fact the values of the pseudo-variables *sender, server*, *self* and *inner* can be accessed through the *message* pseudo-variable as well. For convenience these much-used values are provided as pseudo-variables.

Default Behaviour of Objects

Typically, in most object-oriented languages every class inherits -either directly or indirectly- some default behaviour from a root class called *Object*[4]. The Sina system contains a primitive class called Object which provides a number of standard operations. Typical examples of standard operations used in this thesis are equal, print and copy.

The following actions must be taken to realise such default behaviour: Add an internal, e.g. default, of class Object. All the default properties are defined by class Object. Apart from the standard methods, this may also include conditions and filters. The programmer may then define an input dispatch filter that makes the methods defined by default available on the interface of the object:

```
internals
    default : Object;
conditions
    default.*;
```

---

[3] Unless the message is dispatched to an object that is not controlled by the current class; this object may be redefined, or replaced with a subclass.

[4] Some languages such as C++ do not enforce programmers to inherit from a single class. However, even for C++ programmers it is common practice to introduce a base class such as *Object*.

```
inputfilters
    aDispatchFilter : Dispatch = { default.*, .... } ;
```

To relief the programmer from this 'burden', and ensure that all objects support the default behaviour, the compiler can automatically insert these definitions, including the "default.*" filter element in the dispatch filter of the filter set.

In addition, in case no filters are defined at all, the compiler will insert default filters, for both the input and the output filter set:

```
inputfilters
    default.defDispatch;
outputfilters
    default.defSend;
```

The definitions of these filters by class Object are as follows:

```
defDispatch : Dispatch = {default.*, inner.*};
defSend : Send = { [*]* };   // simply send all messages to their respective targets
```

A more detailed definition of class Object and its methods can be found in the appendix.

The automatic insertion of default behaviour is subject to an additional rule: an internal, condition or filter is not inserted when the concerned class already defines such an entity with the same name. For example, to replace class Object with another class, an internal default can be defined as an instance of the desired class. This will avoid the default insertion of an internal, but not of the other code that is inserted by default.

Atomic Delegations

The mechanism of transactions is a well-known, high-level technique for ensuring both intra- and inter-object consistency. Transactions attempt to maintain system consistency by dealing with possible problems due to exceptions, system failure, and multiple concurrent activities, and in most systems guarantee the permanence of the updates made during a transaction as well.

According to [Haerder 83], a transaction mechanism must provide these four properties: *atomicity*, *consistency*, *isolation* and *durability*. These properties ensure that a transaction always yields a consistent and stable state, even in the presence of system and program failure and concurrent access to shared data. Atomicity, consistency and isolation are provided by the mechanism of atomic delegation [Aksit 91]. Durability is here separated from transactions, and provided through object persistence. We will not discuss object persistence here.

Transactions provided by databases are typically defined in some query language, for a sequence of database operations. Languages such as Argus [Liskov 87] and Avance [Björnerstedt 88] support transactions, which are called atomic actions, as a general mechanism in the language for preserving consistency of concurrently accessed resources.

Most object-oriented systems provide transactions for a program block by delimiting it with constructs like *begin-transaction* and *end-transaction*, or by making the complete method body atomic. This mechanism does not provide integration with object-oriented constructs such as inheritance. This is because combining inherited methods within an atomic construct requires -in the extreme case- the separate declaration of all atomic method combinations, which is not feasible.

Atomic delegation[5] combines the concepts of delegation and atomic action in a uniform model which supports open-endedness of atomic actions. Atomic delegation allows an object to delegate a sequence of messages to one or more designated objects as a single atomic action; such atomic actions are indivisible and recoverable. This mechanism allows the programmer to define classes of atomic actions rather than defining each atomic action separately. Construction of open-ended systems is supported because new atomic actions may be added or existing ones may be modified by changing the delegation relationships between objects without requiring any redefinition of atomic actions, or recompilation of the objects performing the atomic actions.

We now give an example of atomic delegation. Assume that the (paranoid) employer in our example wants to verify all the service requests that clerks receive. This would be expressed as follows:

```
class VerifiedClerk interface
    comment "This is a subclass of clerk, all the operations of clerk must first be verified";
    internals
        clerk : Clerk;  // to inherit from clerk
    externals
        bigBoss : Boss;  // takes care of verifying requests
    inputfilters
        atomDel : Dispatch = { <bigBoss.verify, clerk.*> };
end // class VerifiedClerk interface
```

In the example, the input filter atomDel specifies that the methods inherited from clerk are only accepted when they are preceded by the message verify, which is delegated to the bigBoss object. Note that this requires the client of the object to send the two messages atomically[6], which is expressed with the same brackets, for instance:

```
<aVerClerk.verify; aVerClerk.schedule>;
```

These two messages are executed atomically; either both messages are executed successfully and commit, or an abort and subsequent roll-back takes place.

Note that the atomDel filter declares a number of atomic transactions; all the combinations of verify and methods of the Clerk class. This can be even extended with the following filter declaration:

```
atomDel' : Dispatch = { <bigBoss.*, clerk.*> };
```

This declares all the combinations of a method of the Boss class and a method of the Clerk class as possible transactions. Extensions to these classes will be supported automatically, supporting the open-endedness of the system. It may be clear that the number of possible method combinations can be quite large, and it would not be feasible to declare all possible transactions separately, as conventional mechanisms would require.

---

5 The term atomic *delegation* is a bit misleading, for instance *atomic dispatch* would be more precise, because the mechanism of atomic delegation is applicable to locally defined, delegated and inherited methods.

6 One of the reasons for this is that for a truly reliable transaction, in particular in a distributed system, the system must know that messages are sent atomically from the point where the invocation is made.

Because our realisation of atomic transactions applies locking of objects, dead-locks may occur. This can be the case when two transactions, directly, or indirectly, want to access an object that is already locked by the other transaction. Such situations cannot be predicted, as they depend on other transactions and the timing, ordering and execution speed of the activities. To resolve such situations, a dead-lock detection algorithm is provided, which restarts one of the transactions that caused the dead-lock.

2.6 Specification of the Object Model

In this section a semi-formal model is specified that partially describes the semantics of the composition-filters computation model. This computation model can be considered as an abstraction of the language Sina that we used in this chapter to demonstrate the features of the composition-filters computation model. The motivation for providing this specification is two-fold: firstly, it gives a precise specification of the computation model, in particular of the composition filters mechanism. The second reason is that in chapter 4 we will use the computation model described here as the target for generating object specifications, in particular synchronisation specifications.

The first subsection gives an abstract syntax of the object model, showing the essential components of the composition-filter model. The second subsection describes the semantics of the model as the effects of sending a message to an object.

2.6.1 The Composition-Filters Object Model (CFOM)

We assume the following abstract syntax for composition filter objects. Note that this is a description of the computation model, not the abstract syntax of a particular language such as Sina! In particular, the object model abstracts from all the features that are provided to satisfy software-engineering requirements. Roughly, this can be seen as the distinction between a theoretical and a practical language.

Object $\stackrel{def}{=}$ *symTable:Dictionary; om:ObjectManager; methods:Dictionary; sig:Signature; outputFilters:FilterSet; inputFilters:FilterSet.*

Dictionary $\stackrel{def}{=}$ *Association*.*

Association $\stackrel{def}{=}$ *key:Identifier; value:Object.*

Method $\stackrel{def}{=}$ *PrimitiveMethod* | *UDMethod* .

PrimitiveMethod $\stackrel{def}{=}$ $Object \times Message \rightarrow Object \times Object$.

UDMethod $\stackrel{def}{=}$ *arguments:Dictionary; implementation:Expression.*

Expression $\stackrel{def}{=}$ *Operation** .

Operation $\stackrel{def}{=}$ *MessInvocation* | *Assignment* .

Assignment $\stackrel{def}{=}$ *target:Identifier; source:ObjectSpec.*

MessInvocation $\stackrel{def}{=}$ *receiver:ObjectSpec; selector:Identifier; arguments:ObjectSpecList.*

Message $\stackrel{def}{=}$ *selector:Identifier; arguments:ObjectList; receiver:Object; sender:Object; self:Object.*

ObjectList $\stackrel{def}{=}$ *Object*.*

ObjectSpecList $\stackrel{def}{=}$ *ObjectSpec** .

ObjectSpec $\stackrel{def}{=}$ *Object* | *Identifier* | *Expression* .

FilterSet $\stackrel{def}{=}$ *Filter** .

Filter $\stackrel{def}{=}$ *type:FilterType; init:FiltElems.*

FilterType $\stackrel{def}{=}$ *DispatchFilter* | *MetaFilter* | *ErrorFilter* | *SendFilter* | ..

FiltElems $\stackrel{def}{=}$ *FiltElem** .

FiltElem $\stackrel{def}{=}$ *cond:Condition; operator:ExclOper; messPart:MessProcs* .

Condition $\stackrel{def}{=}$ *impl:Expression.*
ExclOper $\stackrel{def}{=}$ *Enable* | *Exclusion* .
MessProcs $\stackrel{def}{=}$ *MessProc** .
MessProc $\stackrel{def}{=}$ *matchPart:MessSpec; substPart:MessSpec* .
MessSpec $\stackrel{def}{=}$ *target:FiltTargetSpec; selector:FiltSelSpec* .
FiltTargetSpec $\stackrel{def}{=}$ *Object* | *Wildcard* .
FiltSelSpec $\stackrel{def}{=}$ *Identifier* | *Wildcard* .
ObjectManager $\stackrel{def}{=}$.
OMInstr $\stackrel{def}{=}$ *Continue* | *Dispatch* | *Done* | *Exception* | *Reify* .
Identifier $\stackrel{def}{=}$ *value:**S*.

2.6.2 The Composition Filters Computation Model (CFCM)

In this subsection the denotational semantics of the composition-filters computation model is presented. Again, a number of features are ignored or simplified. Firstly, these semantics only describe the functional aspects of the computation model: the effect of sending a message to an object on the state of that object and the object that is returned as a result. Thus, dynamic issues that will be discussed in the forthcoming chapters, such as concurrency, synchronisation, delays, etc. are *not* described by this model!

Object

The semantics of objects are defined as the semantics of sending a message to an object. The result of this is an updated version of the object (the state of the object can be changed), together with the result of the message. The message is forwarded to the set of input filters; these will evaluate the received message one by one, possibly resulting in the execution of that message.

Object : Message $\mapsto$ *Object* $\times$ *Object*
Object $[\![$ *obj* $]\!]$*(mess)* $\stackrel{def}{=}$ *FilterSet* $[\![$*obj.inputFilters* $]\!]$*(mess, obj).*

Dictionary

This is a storage structure that associates identifiers with values (which are always *Object* compounds in the composition filter computation model). The *Dictionary* compound is a list of *Association* compounds, which contain a *key* field that stores the identifier, and a *value* field that stores the object. The semantics of *Dictionary* are defined to be the value associated with the identifier that is offered as an argument. When the identifier is not found in the list, $\perp$ is returned:

Dictionary : Identifier $\mapsto$ *Object*
Dictionary $[\![$ *dict* $]\!]$*(id)* $\stackrel{def}{=}$
if *dict.EMPTY* **then** $\perp$ **else**
if *dict.FIRST.key=id* **then** *dict.FIRST.value* **else** *Dictionary* $[\![$*dict.TAIL* $]\!]$*(id)* **end**
end .

Method

The semantics of a *Method* compound are defined by its execution: it requires the message that caused the execution and the current object as arguments and returns the updated

object and the result of the method execution. This compound realises the choice between primitive methods and user-defined methods, and calls the appropriate semantic functions:

Method : Message × *Object* ↦ *Object* × *Object*
Method ⟦*method* ⟧ *(mess, obj)* ≝
case *method* **of**
PrimitiveMethod ⇒ *PrimitiveMethod*⟦*method*⟧*(mess, obj)* |
UDMethod ⇒ *UDMethod*⟦*method*⟧*(mess, obj)*
end .

PrimitiveMethod

A primitive method implements basic functions. An example is the manipulation of the basic object types such as integers, reals and strings. Features that are offered by the underlying virtual machine or operating system, such as input and output must be provided as primitive methods as well. In the formal model primitive methods are modelled as functions. It is assumed that these functions are pre-defined. The semantics are then simply the application of the pre-defined function to the arguments.

PrimitiveMethod: Message × *Object* ↦ *Object* × *Object*
PrimitiveMethod ⟦*fn* ⟧*(mess, obj)* ≝ *fn(obj, mess)* .

UDMethod

A User-Defined method consists of a number of expressions, where each expression is an assignment or a message expression. A method takes a number of parameters, which can be referred to in message expressions through identifiers. Therefore the list of argument objects provided by the message is copied into a dictionary that associates the formal parameters with the actual parameters. The function *comb* performs this merge of a dictionary and a list of objects. Obviously, it is assumed that the number of formal parameters (in the dictionary) is equal to the number of actual parameters (the arguments provided by *mess*). Note that the dictionary with the arguments is passed on as an additional argument to the *Expression* function.

UDMethod : Message × *Object* ↦ *Object* × *Object*
UDMethod ⟦ <*args, impl*> ⟧ *(mess, obj)* ≝
Expression ⟦*impl* ⟧*(mess, obj, comb(args, mess.arguments))*
where*comb* ≝ λ *dict, list* •
if *dict.EMPTY*
then <>
else (*dict.FIRST* **except** *value=list.FIRST)* ++ *comb(dict.TAIL, list.TAIL)*
end.

Expression

An expression consists of a series of operations; the semantics of *Expression* evaluates each of these, and returns the result of the last operation in the list. The auxiliary function *evaluate* evaluates a single operation. After that it invokes the next operation, handing it the result of the current operation and the resulting object state. When there are no -more-

operations, the result of the previous operation (*prev*) is returned, together with the updated object (*obj*). Notice the use of the pre-defined object *nil*: an empty list of operations returns the value *nil*:

Expression : Message × *Object* × *Dictionary* ↦ *Object* × *Object*
Expression ⟦*operations* ⟧*(mess, obj, args)* ≝
  evaluate(operations, mess, args, obj, nil)
  where *evaluate* ≝ λ *ops, mess, args, obj, prev* •
    if *ops.EMPTY* **then** <*obj, prev*> **else** *evaluate(ops.TAIL, mess, args, obj', prev')* **end** .
    where <*obj', prev'*> ≝ *Operation*⟦*ops.FIRST* ⟧*(mess, obj, args)*

Operation

There are two kinds of operations: a message invocation, or an assignment. *Operation* dispatches the appropriate semantic function:

Operation : Message × *Object* × *Dictionary* ↦ *Object* × *Object*
Operation ⟦*op* ⟧*(mess, obj, args)* ≝
  case *op* **of**
    MessInvocation ⇒ *MessInvocation* ⟦*op* ⟧*(mess, obj, args)* |
    Assignment ⇒ *Assignment* ⟦*op* ⟧*(mess, obj, args)*
  end .

Assignment

The assignment, just like other operations, returns a tuple consisting of the updated object and the result of the operation. The assignment operation always returns *nil* as its result. Because no assignment can be done to a formal parameter, only the symbol table of the object is searched and modified. The assignment involves the evaluation of the right-hand side of the assignment (the *source*); this is done through the *ObjectSpec* semantic function. The auxiliary function *assign* replaces the value of a variable in a dictionary with a new object. The function returns the modified dictionary.

Assignment : Message × *Object* × *Dictionary* ↦ *Object* × *Object*
Assignment ⟦ <*target, source*> ⟧*(mess, obj, args)* ≝
  < *obj* **except** *symTable=newSymTable, nil* >
  where
    newSymTable ≝ *assign(target, ObjectSpec*⟦*source*⟧*(mess, obj, args), obj.symTable)*
    assign ≝ λ *id, newVal, dict* •
      if *dict.EMPTY* **then** <> **else**
        if *dict.FIRST.key=id*
        then *(dict.FIRST* **except** *value=newVal)* ++ *dict.TAIL*
        else *dict.FIRST* ++ *assign(id, newVal, dict.TAIL)* **end**
      end .

MessInvocation

This effectuates the specification of a message-invocation: the receiver is evaluated, the arguments are evaluated, and a new message is constructed and initialised. When the receiver of the message is defined to be 'inner', then the corresponding method is executed immediately. In all other cases, the new message is evaluated by the set of output filters.

MessInvocation : Message × *Object* × *Dictionary* ↦ *Object* × *Object*

MessInvocation ⟦ <*rec, sel, args*> ⟧*(mess, obj, methodArgs)* ≝
if *rec*='inner'
then *Method* ⟦ *Dictionary*⟦ *obj".methods* ⟧*(newMess.selector)* ⟧ *(newMess, obj")*
else *FilterSet* ⟦ *obj".outputFilters* ⟧*(newMess, obj")* **end** .
where
<*obj', rec'* > ≝ *ObjectSpec* ⟦*rec* ⟧*(mess, obj, methodArgs)* .
<*obj", args'* > ≝ *ObjectSpecList* ⟦*args* ⟧*(mess, obj', methodArgs)* .
newMess' ≝ *Message(selector:sel; arguments:args'; receiver:rec';*
sender:obj"; self:rec').

Message

The semantics of a message are its execution, which is expressed by sending the message to its receiver object. The result is the value returned by the message execution only (this is the second component of the tuple returned by the semantic function *Object*).

Note that we permitted ourselves a rather important omission in this specification: the assumption is that any side-effects resulting from sending the message are effective for the state of the sender object and all its components. This problem could be resolved through the introduction of a global state, which must then be given as an additional argument to (practically) all the semantic functions. In an attempt to keep our specification simple, we opted not to do this.

Message : Object .
Message ⟦*mess* ⟧*()* ≝ *Object* ⟦*mess.receiver* ⟧*(mess) (2)*

ObjectSpecList

This is a list of 'object specifications' (see next item).

ObjectSpecList : Message × *Object* × *Dictionary* ↦ *Object* × *ObjectList*
ObjectSpecList ⟦ *objects* ⟧*(mess, obj, args)* ≝
where *evaluate* ≝ λ *objs, mess, obj, args* •
if *objs.EMPTY* **then** <*obj*, <> > **else** < *obj", denot* ++ *tail* > **end** .
where <*obj', denot*> ≝ *ObjectSpec*⟦*objs.FIRST* ⟧*(mess, obj, args)* .
<*obj", tail*> ≝ *evaluate(objs.TAIL, mess, args, obj')* .

ObjectSpec

An 'object specification' can be either a straightforward object, an identifier that refers to an object, or an expression that results in an object when it is evaluated. These three cases are worked out here, where in case of an identifier, first the list of arguments of the method is searched, and then the symbol table defined for the object.

ObjectSpec : Message × *Object* × *Dictionary* ↦ *Object* × *Object*
ObjectSpec ⟦*od* ⟧*(mess, obj, args)* ≝
case *od* **of**
Object ⇒ <*obj, od*> |
Identifier ⇒ **if** *getArg*≠⊥
then <*obj, getArg*>
else <*obj, Dictionary* ⟦*obj.symTable* ⟧*(od)* > |
Expression ⇒ *Expression* ⟦*od* ⟧*(mess, obj, args)*
end .
where *getArg* ≝ *Dictionary* ⟦*args* ⟧*(od).*

FilterSet

This implements the sequence of filters in a filterset: each filter is evaluated by the object manager. As one of the results, a Boolean (*cont*) is returned which indicates whether the next filter(s) in the set should still be evaluated, or not. In the first case, *FilterSet* is called with the rest of the filters in the set, and the updated message and object as its arguments. In the latter case, the result from the last filter evaluation is returned.

FilterSet: Message × *Object* ↦ *Object* × *Object*
FilterSet ⟦ *fs* ⟧*(mess, obj)* ≝
if *fs*.EMPTY **then** ⊥ **else**
if *cont* **then** *Filterset* ⟦*fs*.TAIL ⟧*(mess', obj')* **else** <*obj', res*> **end**
end
where <*cont, mess', obj', res*> ≝ *ObjectManager* ⟦ *om* ⟧ *(mess, obj, fs*.FIRST*)*

ObjectManager

The object manager implements most of the virtual machine functions, such as message dispatch etc. This enables the OM to keep an up-to-date representation of the current object state (e.g. number of active threads, etc.). The object manager semantics define what to do for a specific filter with a particular message. This will result in a Boolean that defines whether the next filter in the set must be evaluated (e.g. when an Error filter has been passed), or not (e.g. when an error has occurred, or a dispatch has taken place). In addition the modified message, the new object state and the result of the message execution (when applicable) are returned.

ObjectManager : Message × *Object* × *Filter* ↦ 𝔹 × *Message* × *Object* × *Object*
ObjectManager ⟦ *om* ⟧*(mess, obj, filt)* ≝
case *filtAction* **of**
CONTINUE ⇒ <*true, mess', obj,* ⊥ > |
DONE ⇒ <*false,* ⊥*, obj,* ⊥ > |
DISPATCH ⇒ <*false,* ⊥*, obj', result*> **where** <*obj', result*> ≝ *dispatching(mess', obj)* |
SEND ⇒ <*false,* ⊥*, obj, result*> **where** *result* ≝ *sending(mess')* |
EXCEPTION ⇒ <*false,* ⊥*, obj,* ⊥> **where error** *// no result can be returned now..* |
REIFY ⇒ <*true, mess, obj, result*> **where** *result* ≝ *reification(mess, mess')* .
end
where*filtAction* ≝ *FilterType* ⟦*filt.type* ⟧*(mess', acceptance, obj).*
where <*mess', acceptance*> ≝ *Filter*⟦*filt* ⟧*(mess, obj).*
dispatching ≝ λ *message, obj* •
if *message.receiver=obj*
then *Method* ⟦ *Dictionary*⟦ *obj.methods* ⟧*(message.selector)* ⟧ *(message, obj)*
else <*obj, Message*⟦*message*⟧*()* > **end** .
sending ≝ λ *message* • *Message*⟦*message*⟧*()* .
reification ≝ λ *newMess, oldMess* • *Message*⟦*newMess* **except** *args=*<*oldMess*>⟧*()*

Filter

This implements the semantics of a single filter: it evaluates the message for the list of filter elements, and returns the result of this (a Boolean indicating acceptance plus the -possibly modified- message):

Filter: Message × *Object* → *Message* × 𝔹

Filter[[<*t*, *elems*>]](*mess*, *obj*) $\stackrel{def}{=}$
  FiltElems[[*elems*]](*mess*, *obj*)

FilterType

This is the supertype of the various filtertype implementations. It thus implements the choice between the various filtertypes through a case statement:

FilterType: *Message* × $\mathbb{B}$ × *Object* → *OMInstr*
FilterType [[*ft*]](*mess*, *acceptance*, *obj*) $\stackrel{def}{=}$
  case *ft* **of**
    DispatchFilter ⇒ *DispatchFilter* [[*ft*]](*acceptance*, *mess*, *obj*) |
    ErrorFilter ⇒ *ErrorFilter* [[*ft*]](*acceptance*, *mess*, *obj*) |
    MetaFilter ⇒ *MetaFilter* [[*ft*]](*acceptance*, *mess*, *obj*) |
    SendFilter ⇒ *SendFilter* [[*ft*]](*acceptance*, *mess*, *obj*) |
    ....
  end

DispatchFilter

This implements the *Dispatch* filter: when the message is accepted, then the OM instruction DISPATCH is the result, otherwise it is CONTINUE.

DispatchFilter: $\mathbb{B}$ × *Message* × *Object* → *OMInstr*
DispatchFilter [[*df*]](*acceptance*, *mess*, *obj*) $\stackrel{def}{=}$
  if *acceptance* **then** *DISPATCH* **else** *CONTINUE* **end**

ErrorFilter

This implements the *Error* filter: when the message is accepted, then the OM instruction CONTINUE is the result, otherwise it is EXCEPTION.

ErrorFilter: $\mathbb{B}$ × *Message* × *Object* → *OMInstr*
ErrorFilter [[*df*]](*acceptance*, *mess*, *obj*) $\stackrel{def}{=}$
  if *acceptance* **then** *CONTINUE* **else** *EXCEPTION* **end**

MetaFilter

This implements the *Meta* filter: when the message is accepted, then the OM instruction REIFY is the result, otherwise it is CONTINUE.

Note that, although we give the semantics of the MetaFilter, these semantics do not describe the semantics of firing and continuing messages. This is a part of the primitive object type *message*. It requires the addition of context information to the *Message* compound, though, because after a fire it must be known where (at which object and filter) to continue the message.

MetaFilter: $\mathbb{B}$ × *Message* × *Object* → *OMInstr*
MetaFilter [[*df*]](*acceptance*, *mess*, *obj*) $\stackrel{def}{=}$
  if *acceptance* **then** *REIFY* **else** *CONTINUE* **end**

SendFilter

This implements the *Send* filter: when the message is accepted, then the OM instruction SEND is the result, otherwise it is CONTINUE.

SendFilter: $\mathbb{B}$ × *Message* × *Object* → *OMInstr*
SendFilter [[*df*]](*acceptance*, *mess*, *obj*) $\stackrel{def}{=}$
  if *acceptance* **then** *SEND* **else** *CONTINUE* **end**

Other types of filter are not described here, but can be defined analogously.

FiltElems

The evaluation of the sequence of list elements results in a tuple consisting of a message and a Boolean, where the latter indicates message acceptance. If a filter-element is encountered that accepts the message, the -possibly substituted- message is returned, and no further elements are checked. If no message element matches, the original message is returned, along with the Boolean *false* to indicate that the message is not accepted:

$FiltElems : Message \times Object \rightarrow Message \times \mathbb{B}$

$FiltElems \llbracket\ elems\ \rrbracket (mess, obj) \stackrel{def}{=}$
  if $elems.EMPTY$ **then** $\langle mess, false \rangle$ **else**
    if *accept* **then** $\langle mess', true \rangle$ **else** $FiltElems \llbracket\ elems.TAIL\ \rrbracket (mess, obj)$ **end end**
  where $\langle mess', accept \rangle \stackrel{def}{=} FiltElem \llbracket\ elems.HEAD\ \rrbracket (mess, obj)$

FiltElem

For each filter element, first the condition is evaluated; if it is true, then it is tried to match the right-hand side of the filter element, otherwise the original message and the Boolean *false* is returned, indicating that the filter element does not accept the message:

$FiltElem : Message \times Object \rightarrow Message \times \mathbb{B}$

$FiltElem \llbracket\ \langle cond, oper, messpart \rangle\ \rrbracket (mess, obj) \stackrel{def}{=}$
  if $Condition \llbracket\ cond\ \rrbracket (mess, obj)$
  then $ExclOper \llbracket\ oper\ \rrbracket (mess, MessProcs \llbracket\ messpart\ \rrbracket (mess))$
  else $\langle mess, false \rangle$ **end**

Condition

A condition is an expression without side-effects that returns a Boolean value. The conversion from an *Object* to an element of $\mathbb{B}$ is performed by the function *convertObject2Boolean*. The implementation of this function depends on how booleans are represented as *Objects*. We do not bother with this, and leave the implementation of *convertObject2Boolean* open. Note that the semantic function *Expression* expects as an argument a dictionary with Identifier-Object associations. In order to fulfil this, an empty *Dictionary* is provided:

$Condition: Message \times Object \rightarrow \mathbb{B}$

$Condition \llbracket\ \langle impl \rangle\ \rrbracket (mess, obj) \stackrel{def}{=}$
  $convertObject2Boolean(Expression \llbracket\ impl\ \rrbracket (mess, obj, Dictionary(\langle\rangle))).$
  where $convertObject2Boolean : Object \rightarrow \mathbb{B}$

ExclOper

This is the operator that separates the condition from the message processing part. It can be either '=>', which is the enable operator, or '~>', which is the exclusion operator. Depending on the type of operator, the result of the matching phase can be affected, and the new message is either adopted or discarded:

ExclOper: Message × Message × IB → Message × IB
ExclOper ⟦ *oper* ⟧*(mess, newmess, accept)* ≝
case *oper* **of**
Enable ⇒ **if** *accept* **then** <*newmess, accept*> **else** <*mess, accept*> |
Exclusion ⇒ < *mess,* ¬*accept* >
end .

MessProcs

This has semantics similar to the *FiltElems* compound: it iterates over the elements in the list until the evaluation of an element results in acceptance. The result that is returned is a tuple consisting of the acceptance Boolean and the (optionally) modified message.

MessProcs: Message → Message × IB
MessProcs ⟦ *procs* ⟧ *(mess)* ≝
if *procs.EMPTY* **then** <*mess, false*> **else**
if *accept* **then** <*mess', accept*> **else** *MessProcs*⟦ *procs.TAIL* ⟧*(mess)* **end end**
where <*mess', accept*> ≝ *MessProc*⟦ *procs.HEAD* ⟧*(mess)*

MessProc

This implements an important part of the filter mechanism; the matching and substitution of messages. Two different situations are distinguished: the first is when there is no matching part (between the square brackets) defined. In this case matching -and thus, acceptance- is based on matching the selector and the signature of the target. Otherwise, the specification of the matching part (of both target and selector) is tried to match against the receiver and selector of the message:

MessProc : Message → Message × IB
MessProc ⟦ < <*matchTar, matchSel*>, <*substTar, substSel*> > ⟧ *(mess)* ≝
if (*matchTar*=⊥) ∧ (*matchSel*=⊥)
then if *matches(message.selector, matchSel)* ∧ *signMatches(substTar, message.selector)*
then < *substTarget(substTar)* ∘ *substSelec(substSel)(mess), true*>
else <*mess, false*> **end**
else if *matches(message.receiver, matchTar)* ∧ *matches(message.selector, matchSel)*
then < *substTarget(substTar)* ∘ *substSelec(substSel)(mess), true* >
else <*mess, false*> **end**
end
where
signMatches ≝ λ *tar, sel* •
case *tar* **of** *Wildcard*⇒*true* | *Identifier*⇒*Signature*⟦ *tar.sig* ⟧*(sel)* **end**.
substTarget ≝ λ *mess, tar* •
case *tar* **of** *Wildcard*⇒*mess* | *Identifier*⇒(*mess* **except** *receiver=tar)* **end**.
substSelec ≝ λ *mess, sel* •
case *sel* **of** *Wildcard*⇒*mess* | *Identifier*⇒(*mess* **except** *selector=sel)* **end**.
matches ≝ λ *pattern, match* •
case *pattern* **of** *Wildcard*⇒*true* | *Identifier*⇒(*pattern=match)* **end**.

Signature

The semantics of the signature are those of inclusion: when the argument can be found in the list of selectors, *true* is returned, otherwise *false*:

Signature : Identifier $\rightarrow$ $\mathbb{B}$

Signature $[\![$ *list* $]\!]$*(selector)* $\stackrel{def}{=}$

over *list* **apply** λ *elem* • *elem=selector* **combine** $\vee$ **empty** *false* **end**

2.7 Discussion

This section evaluates the composition filters approach that has been presented in this chapter. First some related work is described, then the properties of the composition filters model are summarised.

2.7.1 Related Work

The distinction between the composition-filters model and the conventional object models should be rather clear, as this is basically described by the interface part of a composition filters object: the kernel object, or implementation part, adheres largely to the conventional object model, and the extensions to this model are described in the interface part. Inheritance is an exception to this; the kernel object model is object-based and does not support this mechanism. In our model, inheritance is realised in the interface part of an object.

Thus, we do not further compare with programming languages that adopt the conventional object model, such as Smalltalk [Goldberg 83], Eiffel [Meyer 88,92] and C++ [Stroustrup 86], but discuss a few languages that provide specific constructs for defining the interface of objects, and the manipulation of received messages. As far as these languages provide facilities for concurrent programming, we mostly omit these in this discussion. In the next chapter the support for concurrency and synchronisation in object-oriented languages is discussed.

Concurrent Aggregates

This language, described in [Chien 91] and [Chien 93], is aimed at the programming of fine-grain parallel architectures (massive parallelism). It is concerned with deriving a sufficient number of concurrent activities from a program, and with managing the complexity in writing large programs.

The aggregate construct is motivated by the fact that most concurrent object-oriented programming languages serialise messages to nested parts. The language focuses on the use of aggregates to represent collections of objects. Four additional concepts support this: intra-aggregate addressing, which enables the parts in an aggregate to access other parts. Delegation is provided to support the composition of the behaviour of an aggregate from other aggregates. This is somewhat similar to the functionality of a dispatch filter.

First class representations of messages are provided, allowing the software developer to write abstractions that manipulate messages, which can for instance be used to implement control structures. Finally, Concurrent Aggregates treats continuations as first-class objects, and user-defined objects can be used as continuations. This supports the construction of various synchronisation techniques. The latter two concepts are comparable to the application of ACTs in Sina.

MAUD

The MAUD language [Frølund 93] is related to Sina mainly because it provides a mechanism for intercepting incoming and outgoing messages. Each object in MAUD owns three meta-objects called a *dispatcher*, a *mail queue* and *acquaintances*. The sent and received

messages are handled by the dispatcher and mail queue objects respectively. The acquaintances object contains a list of objects that may be addressed by its owner object. With MAUD, one can implement coordinated behaviour by replacing the meta-objects with the objects implementing the required protocol. To install a protocol for an object the original mail queue and dispatcher must be replaced by a pair implementing the required protocol.

In MAUD a shared protocol among objects is divided into mail queue and dispatcher objects which are created separately for each object. An important difference with the filter mechanism is that filters provide a consistent framework for manipulating incoming messages, with predefined filter types for common problems. Application-specific message manipulation problems and coordinated behaviour are solved through ACTs.

Procol

This concurrent object-based language ([Bos 89], [Laffra 92]) provides a mechanism called *protocol* that defines the interaction protocol between the sender and receiver of a message. Further properties of the language are that it provides concurrency, delegation, persistence and a constraint mechanism.

The protocol of an object describes interaction patterns with extended regular expressions. Received messages are matched based on the sender of the message (or its type), the message selector, the arguments of a message and the current state of the receiving object. The latter appears in two forms: firstly the sequence (history) of received messages determines the current state of the protocol. This determines the one -or more- messages that are expected next according to the regular expression. Secondly the regular expressions can be augmented with guards, which are boolean expressions on the internal state of the object, such as the values of instance variables.

Protocols serve the following purposes [Laffra 92]:

- ❏ They serve as an interface specification for other objects.
- ❏ They sequence interactions between objects.
- ❏ They control access to the methods of the object.
- ❏ They perform type or identity checking on clients.
- ❏ They function as a composition rule, since they can specify relations with client objects, and can delegate requests to other objects.

In the next chapter the role of protocols in concurrency will be discussed. The constraint mechanism provides some support for specifying coordinated behaviour. It cannot deal with first-class message representations, though.

Encapsulators

The e*ncapsulators* framework [Pascoe 86] offers an approach that is similar to the ideas motivating the composition-filters model: an application object can be surrounded with a layer that intercepts messages that are sent to the object and the replies of those messages. The encapsulators are special objects that implement this layer. An encapsulator defines a pre-action, that is executed upon message reception, and a post-action, that is executed when the result of the message is returned.

Encapsulators are implemented as an extension to the Smalltalk-80 system. Applications of the encapsulators mechanism that are discussed in [Pascoe 86] are a *Monitor*, which enforces mutual exclusion for Smalltalk objects, an *Atomic* encapsulator that offers a simple roll-back mechanism for message execution, and a *Model*, that generates triggers when certain messages are executed by the object.

One of the differences with the composition-filters model is that the encapsulators do not associate actions with messages that are sent, whereas the composition-filters model does not associate actions to replies. The actions in encapsulators are straightforward Smalltalk method implementations. The composition-filters model aims at providing abstractions to manage complex object behaviour. In addition, the composition filters are also used to express data abstraction techniques, whereas encapsulators do not deal with this.

Contracts

In the area of object-oriented modelling, the idea of specifying object interactions as an explicit module is applied by *contracts* (defined in [Helm 90] and [Holland 92]). Contracts are used to specify the *contractual obligations* that a set of *participants* must satisfy. It is possible to *refine* a contract in order to make it more specific and it is possible to *include* existing contracts in a new contract. In its first version [Helm 90] a declarative language was introduced to define contractual obligations. In the second version [Holland 92], however, a procedural language was adopted instead of a declarative one. In the following sections we refer only to the second version of contracts.

A contract specification includes the specification of the participating objects, the contractual obligations of all participants, the invariants to be maintained by the participants and the method which instantiates a contract.

A contract can be seen as an *abstract class*, defining both abstract and concrete methods for its participants. The abstract methods must be provided by the participants themselves. The concrete methods of the contract (or its refinement) override the concrete implementations of the participants. A contract may also define variables that are shared by all the participants. In order to put a contract to use, a conformance declaration must be made which initialises the contract with actual participants. Obviously, these participants have to satisfy the contractual obligations of the contract. An object may participate in several contracts. Contracts offer two alternatives: either the methods are implemented at the contract specification, or they are distributed over the participating classes.

Contracts are primarily targeted as a design tool. Contracts are quite useful for the implementation of coordinated behaviour and the abstraction of object interactions but are unable to reflect upon the actual message interactions between objects for purposes such as monitoring, synchronising and manipulating messages.

Conclusion

The most important distinction between the languages and systems discussed here and the composition-filters model is that the latter is a framework that offers limited reflection on messages in a declarative way and with a consistent notation, while offering open-endedness so that it can be applied in a range of application-domains. This framework takes the place

of numerous language constructs that would be required to offer the same functionality in a conventional approach.

2.7.2 Evaluation

We summarise the most important properties of the composition filters model and Sina:

- The composition-filters object model is a modular extension of the conventional object-based model. This also separates the implementation of an object from its interface specification.
- The model provides strong encapsulation; even subclasses cannot access the implementation aspects of their superclasses.
- Interaction between objects is based on a single, request-reply model of communication.
- Strong typing is supported by the language, based on the signature of objects, and thereby independent of the inheritance hierarchy.
- Several variations of data abstraction techniques are supported. The behaviour of an object is defined as the composition of the behaviour of its internal or external components.
- The filters completely control the externally visible behaviour of objects as they can check and manipulate the incoming and outgoing messages.
- The filter framework offers a consistent notation and model for specifying the properties of an object. Filter types offer an open-ended solution for incorporating modelling techniques from a range of different domains in a single system.
- The various techniques that are offered by the filters for solving problems in a variety of domains are *orthogonal*, which means that they can be composed without interfering.
- Abstract communication types provide the software developer with a tool for defining abstractions that manipulate messages as first-class objects.

The filter mechanism provides tailored, declarative, reflection capabilities on messages with a granularity and specification technique that supports the management of complexity in large programs. In addition, the filter declaration is a specification of the interface the object offers to other objects: it can be considered as a *contract* for the clients of the object [Meyer 88].

In this chapter we discussed three filter types: *Error*, *Meta* and *Dispatch*. These filter types can be used to accomplish the following techniques:

- *Multiple views*: the external interface of an object may differ per client object or due to the state of the receiver object or the system.
- *Assertions*: a limited form of assertions, or preconditions, is possible by associating conditions with messages. This can be applied to received as well as sent messages.
- *Data abstraction techniques*: the dispatching mechanism supports a variety of data abstraction techniques. It allows for expressing both inheritance and delegation, in single and multiple forms, and with the possibility of dynamically disabling and enabling inheritance and delegation relations. All these variations can be freely mixed. Naming conflicts can be resolved through the ordering in filter definitions or through renaming of messages.

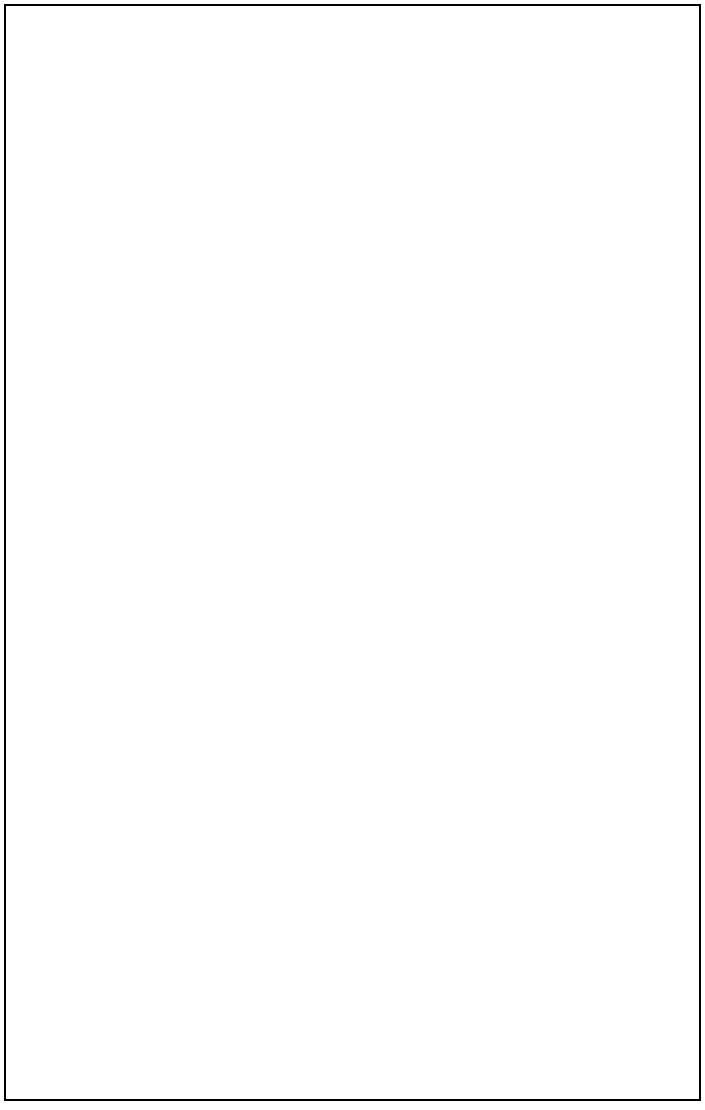
- *Atomic transactions*: the specification of atomic transactions is integrated with the object-oriented model. It supports the atomic execution of combinations of locally defined and reused methods without a combinatorial explosion of the number of transaction definitions.
- *Evolving behaviour*: dynamic inheritance allows for the behaviour of an object to change during its life-time, according to the particular circumstances. This means that the methods that are visible on the interface of the object may vary.
- *Alternative implementations*: the notion of alternative implementations is a special case of dynamic inheritance. It means that the interface of an object does not change, but the same message can be implemented by and inherited from a different parent.
- *Coordinated behaviour*: ACTs can be used to define coordinated behaviour between objects. This allows for monitoring, checking, controlling and managing the messages that are interchanged between objects. ACTs are fully integrated first-class objects, and thus have all the expressive power, reusability and extensibility properties of the composition filters object model.
- *User-defined message passing semantics*: ACTs can also be applied for manipulating the semantics of message passing. This may be used for modifying message contents, for changing the semantics to multi-cast or broadcasts, or for changing the synchronisation properties of message sends.

A range of variations and techniques with composition filters can be imagined, have been investigated already, or are the subject of current research. These include:

- The provision of a query mechanism that is integrated within the object-oriented composition-filters model (described in [Aksit 92a]).
- The combination of the query mechanism and inheritance leads to the notion of *associative inheritance*, which has been published in [Aksit 92a] as well.
- The reusable and extensible specification of real-time constraints is described in [Sterren 93] and [Aksit 94b].

Further, the concept of composition filters can be fruitfully applied to reusable and extensible synchronisation specifications. This is the topic of the forthcoming chapter.

Chapter 3



Concurrency and Synchronisation

3.1 Introduction and Background

This chapter deals with the issues involved in introducing concurrency in the composition-filters model. To coordinate the concurrent activities, a mechanism for synchronisation is required. In this chapter we investigate the appropriate techniques for creating and controlling concurrency. Our prime concern in this investigation is to find mechanisms that retain the typical object-oriented characteristics of encapsulation, reusability, extensibility and maintainability.

In this section we explore the reasons for introducing concurrency, concurrent object models and various message passing semantics. Then we give a brief overview of the various approaches to synchronisation in object-oriented models. Finally we define a number of criteria for effective concurrent object-oriented programming languages.

3.1.1 The Need for Concurrency.

Our most important motivation for this research derives from the goal of object-oriented software development: to construct a model of the real world. We do this because we want to build a conceptual model, or a description, of the real world. We may also do this because we want to perform some computation, adopting the *computation-through-simulation* paradigm. A closer look at the real world reveals that concurrent activities appear everywhere. According to Wegner: 'The real world is concurrent rather than sequential.' [Wegner 91]. In the object-oriented paradigm a program is a collection of objects that all represent a physical or conceptual entity in the real world. Each of these entities may represent or contain one or more activities [Yonezawa 87].

The entities in the real world are often nested, and so are the objects in our system. Each of these nested objects may encapsulate nested activities again, and multiple interactions may occur between these objects in parallel. For example, consider a bank with a number of clerks serving customers. Provided it is open, customers may enter the bank and wait in line until they can interact with one of the clerks at the counter. The clerks may in turn interact with other employees or departments of the bank in order to fulfil the requests of the customers. This is illustrated by figure 3.1.1.

According to the object-oriented paradigm, each object is an autonomous agent, capable of handling received requests. *Active objects*[1] are objects that are capable of controlling and scheduling the received requests before they are served by the object [Nierstrasz 91].

This observation leads to the following conclusions: to properly model real-world situations, concurrent activities must be modelled [Wegner 91]. Each object must be capable of dealing with multiple, concurrent requests. Requests may be serialised or trigger

[1] In the literature, the term *active objects* is sometimes used to denote the association of a process with an object. This is *not* what we mean with active objects! At a certain point in time, no activities may be performed by an object at all, whereas at another time it performs several activities at once. The distinction between active objects and passive objects is that the latter have no control over the activities that they perform.

concurrent activities *within* the object. Objects can be nested, even when they encapsulate or serve -concurrent- activities. In summary, we demand *active* objects, that allow *intra-object concurrency*, and support *nesting* of active objects.

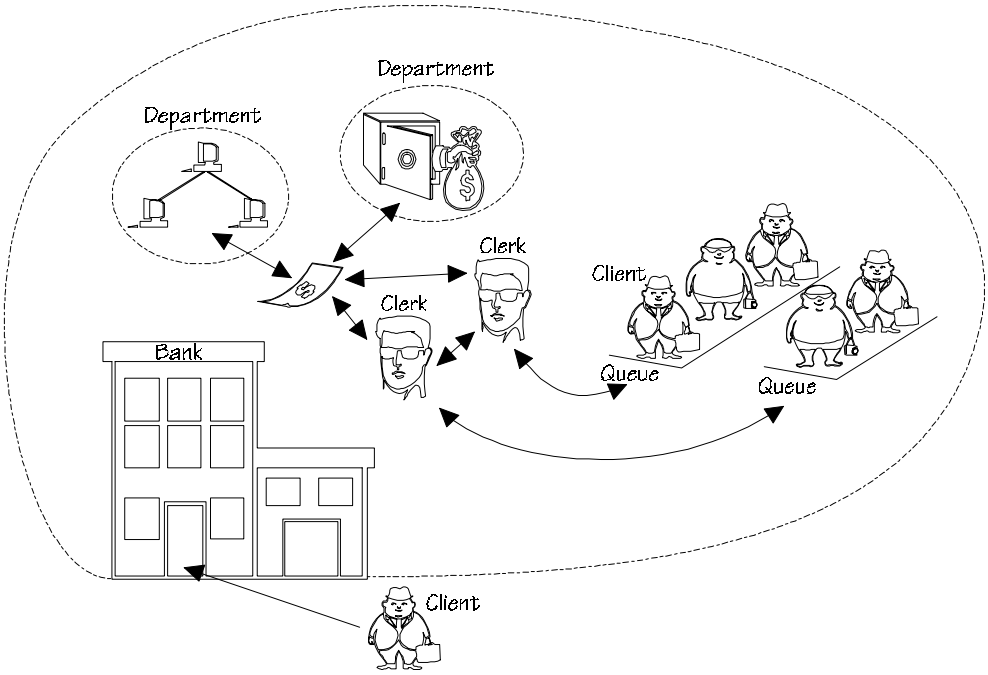


Figure 3.1.1 An example of a real world model showing nested active entities.

As stated before, our primary motivation for adopting concurrency lies with the modelling issues that were just discussed. It should be remarked, however, that an object-oriented model lends itself much more for a realisation on a distributed or parallel architecture, because objects are concurrent *by nature*. It is thus relatively easy to identify tasks that can be executed in parallel [Agha 90].

3.1.2 Models of Concurrency

There are many alternative approaches for introducing concurrency and synchronisation in an object-oriented language. In this subsection we categorise these approaches. The categorisation leans on the classifications made in [Wegner 92] and [Papathomas 91].

Module of Synchronisation:

We previously discussed objects that handle the synchronisation of messages at the object boundary. This is not the only level for applying synchronisation, though. We distinguish two categories:

- *code level*: In this category, messages are always accepted for execution. Concurrency is controlled through conventional mechanisms such as semaphores and monitors, which appear as statements that are embedded within the implementations of method bodies. This category is referred to as *passive* objects and is exemplified by Smalltalk-80 [Goldberg 83], Emerald [Black 86] and Trellis/Owl [Moss 87]. The internal state of

objects is only protected against inconsistencies when the methods that affect the state contain synchronisation statements.

- *object level*: In this case we speak about *active* objects: upon reception, messages can be delayed until apt for execution. Thus, synchronisation occurs at the object boundary, thereby protecting the internal consistency of objects. Various types of synchronisation mechanisms can be applied at this level. Examples of languages that synchronise at the object level are POOL [America 87, 90], Procol ([Bos 89] and [Laffra 92]) and Guide [Decouchant 91].

Of these two approaches, synchronisation at the object level is preferred, as it more closely reflects the notion of autonomous and *self-sufficient* objects in modelling.

Homogeneity of Synchronisation

A number of systems distinguish between synchronised and unsynchronised objects, or *active* and *passive* objects. We call this a *heterogeneous* approach. In the *homogeneous* approach this distinction is not made. Passive objects cannot be protected against simultaneous access[2]. Some languages require passive modules to be encapsulated by an active module, which then becomes responsible for its protection. Examples of languages that adopt a heterogeneous approach are Argus [Liskov 87], ACT++ [Kafura 89,90] and Eiffel// [Caromel 90].

A homogeneous, object-level approach is preferable; because it makes the object the granularity of concurrency control, each object becomes less dependent of a particular context, thus improving reusability. The heterogeneous approach seems to be motivated mainly by an efficiency argument. The following table shows a number of examples for each category:

Synchronisation	at the object level	at the code level
homogeneous	POOL, Procol, Guide, Sina/st	Smalltalk, Trellis/Owl, Emerald
heterogeneous	ACT++, Eiffel//	Argus

Granularity of Concurrency

One approach to promoting the consistency and safety of the object is by reducing the amount of concurrency that is allowed: large-grained concurrency can protect groups of objects to keep them in a consistent state. For example, consider a collection of account objects in a banking system: when only a single thread can execute within this collection, transferring money from one account to another can be done safely. If multiple concurrent threads can be active within the collection, it is more difficult to maintain consistency, in particular when they access the same objects.

---

2 Note that we consider only the synchronisation of objects, in hybrid languages that support both objects and conventional data types, the latter are always passive.

Following [Wegner 92] we distinguish the following levels of granularity of concurrency in object-oriented systems, from large to fine-grained:

- *mutual exclusion*: This means that the object or module allows only a single thread to be active within it. Other (possibly pending) requests are only allowed after the current request is completely finished. Examples of this approach are objects in ABCL, POOL and Orca [Bal 92].
- *quasi-concurrency*: A quasi-concurrent object is one that allows only one thread to be active at a time, but once that thread becomes blocked, another thread may become active; threads can be interleaved. This approach is adopted by monitors [Hoare 74] and domains in Hybrid [Nierstrasz 87].
- *internal concurrency*: If multiple threads can be active at the same moment within an object, we term this *intra-object concurrency*. This allows for very fine-grained concurrency, and is supported by e.g. Guide [Decouchant 91], Sina/st [Tripathi 88] and Parallel Objects [Corradi 91].

Internal concurrency is desirable, because it allows for more effective real-world modelling. Intra-object concurrency is also desirable for reusability reasons; nested activities can be modelled without breaking encapsulation [Nierstrasz 91]. In addition, sequential and quasi-concurrent objects have a serialising effect in object-oriented programs, which is in particular severe in the case of hierarchically nested object structures. This serialisation can have a negative influence on performance in implementations on parallel architectures [Chien 91], as it reduces the amount of concurrent threads in an application.

> In [Nierstrasz 91] and [Papathomas 91] a distinction is made between controlled and unconstrained internal concurrency. Controlled internal concurrency means that an object itself can control and limit the amount of internal concurrency, whereas in the case of unconstrained internal concurrency this is not the case. As this can expressed as the distinction between passive and active objects, we do not treat it separately.

In summary, we have stated in this subsection that the preferred module of synchronisation is at the object level, that all objects in a system should be capable of determining their own synchronisation conditions, and that allowing intra-object concurrency improves modelling, reuse and performance properties.

3.1.3 Message Passing Semantics

Since objects can only interact by sending messages, message passing is an important means for creating concurrency and for synchronisation among objects. The invocation of a message incorporates many aspects: it interrupts the current thread of control, involves a synchronisation with the receiver of the message, and triggers an execution at the receiver object. In addition, the message invocation may cause a reply to be sent back to the sender of the message.

We distinguish two fundamental one-way message-sending constructs: *asynchronous* respectively *synchronous* message passing[3]. In the case of asynchronous communication the

[3] The terms synchronous and asynchronous are sometimes used differently, e.g. to denote a request-reply model respectively a one-way message passing model.

sender object continues its activities by executing subsequent statements after sending the message. It does not matter whether the receiver object is ready to communicate or not. In case the receiver is not ready, a buffering mechanism is required. When the receiver is -or becomes- ready, the resulting execution is performed in parallel with the activity of the sender object. This is illustrated by the following figure. It is an event diagram that shows the execution flow of two objects, sender object A and receiver object B. Time progresses from the top to the bottom of the lines. The thick lines designate activities, or processes[4].

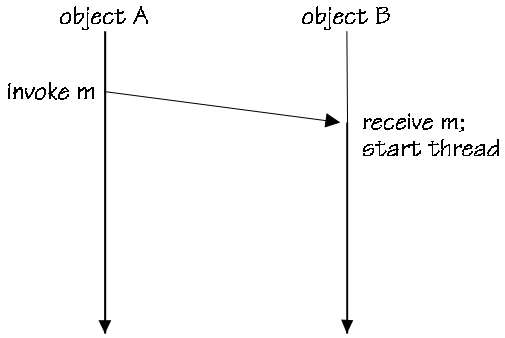


Figure 3.1.2 Asynchronous message passing.

An asynchronous one-way message-passing model is applied in actor-based models such as ABCL/1 and Act-1 [Lieberman 87], and in systems such as SR [Andrews 81], Eiffel// and Parallel Objects [Corradi 91].

In the case of synchronous message passing, the sender object is blocked after initiating a message invocation until the receiver object is ready to communicate. When this is the case, both the sender and the receiver object continue executing in parallel. This is illustrated by the following figure, where the "■" symbol designates that the corresponding thread is blocked:

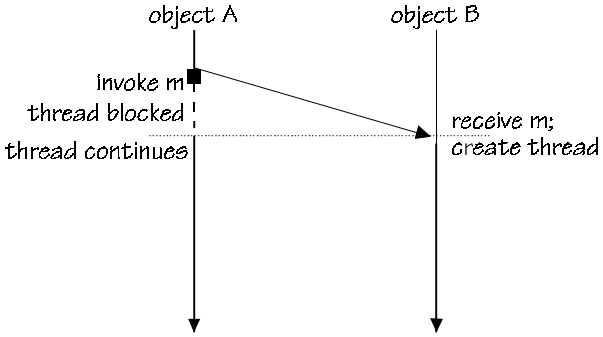


Figure 3.1.3 Synchronous message passing

The synchronous message passing model is used in Communicating Sequential Processes (CSP) [Hoare 78] and Procol [Bos 89]. This mode of communication does not require any message buffering.

We continue our discussion with a number of *request-reply* based message passing models. These are based on the conventional function call in procedural languages. The

4 The beginning of a thick line can, but does not necessarily, mean the creation of a new process. We abstract here from these issues: a message invocation could also awake a blocked process.

implementation of such languages on distributed architectures led to the notion of the *Remote Procedure Call* (RPC, [Nelson 81]), which simulates a conventional procedure call. The main difference is that the sender and receiver object do not necessarily execute within the same process or physical processor:

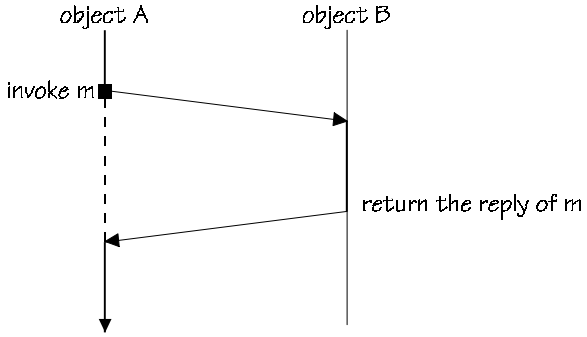


Figure 3.1.4 (Blocking) Remote Procedure Call

The client object completely blocks its execution until it receives the reply to the sent message. Therefore this message passing model is also referred to as *blocking RPC*. At all times, there is only a single thread of control active. This model has been applied in Ada [Ada 80], Argus, DP [Brinch-Hansen 78], POOL, and SR.

A variation to the blocking RPC is the *non-blocking RPC*[5]. This is similar to the blocking RPC, except that the receiver object may continue executing after it has sent the reply. In this case the sender and receiver object execute in parallel, but only after the reply has been received by the sender object:

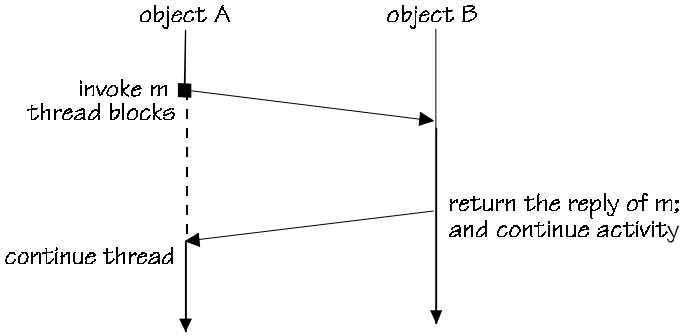


Figure 3.1.5 Non-blocking RPC

Returning a reply while continuing execution is also referred to as an *early return*. The advantage of this model is that it offers a conventional RPC protocol to the client (i.e. the sender object), whereas the server may determine to create additional parallelism. This mechanism has been adopted by Ada and POOL (the so-called *post-processing section*).

Another variation to the RPC model is the so-called *future RPC*. This model retains the basic RPC semantics, while allowing additional parallelism. The model lets the sender object send an asynchronous request to the receiver object. The sender object will then continue

5 In the distributed systems domain the term *non-blocking RPC* is commonly used to designate RPC calls with semantics that are similar to asynchronous message passing; the caller can continue its activity immediately after making the invocation.

execution, up to the moment that the result of the invocation is required for further processing. When the receiver object has already returned the reply, the sender may immediately continue its execution, but when the reply has not been received yet, the sender thread is blocked. The thread will continue its execution when the reply is received. This is illustrated by figure 3.1.6.

The future mechanism is realised by introducing a *proxy* object, which is returned after the message request, to replace the reply object. When the reply has not been returned yet, all requests to the proxy are blocked. Once the reply has been returned, the proxy provides access to the returned object. Futures are supported by for instance ABCL/1 and ConcurrentSmalltalk [Yokote 87a, 87b].

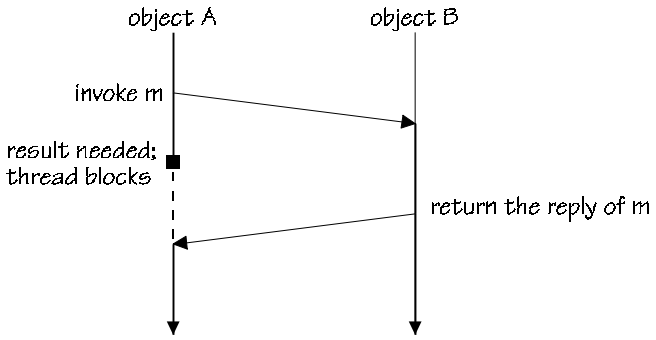


Figure 3.1.6 Future RPC

Other message passing semantics include multi-casts (e.g. in ABCL/1) and coordinated termination (e.g. in ABCL/1 and Orient84/K [Ishikawa 87]). A multi-cast means that the same message is sent in parallel to several receiver objects. Coordinated termination means that the sender is blocked until either one of the replies (*or* synchronisation) or until all replies (*and* synchronisation) have been received.

In general the RPC-style of communication is attractive as it offers a protocol with a higher level of abstraction between the client and server [Papathomas 91]. In particular the association of a specific reply with the correct request is a problem in one-way message passing models. An important motivation for the application of one-way message passing models is that they increase the amount of concurrency in the application.

The one-way message passing models can be used to construct higher-level message passing models, though. Similarly, in [Yonezawa 87] the future RPC and asynchronous message passing are shown to be expressible in terms of the blocking RPC. The choice of message passing model(s) to apply in a system or language thus depends on the desired level of abstraction, the efficiency of the individual constructs and the amount of concurrency they introduce.

3.1.4 An Overview of Synchronisation Schemes.

In the recent years a tremendous amount of concurrent object-based and object-oriented programming languages have been developed[6]. Several publications give an overview of the

6 A quick survey of publications showed over 50 different proposals for concurrent object-based models.

relevant concepts and languages, for example, [Tomlinson 89b], [Agha 90], [Wyatt 92] and [Papathomas 91]. In [Nierstrasz 91, 93a] and [Matsuoka 93a] more analytical studies of concurrent object-oriented languages are presented.

In this subsection we will briefly discuss a selection of relevant language proposals, which is by no means complete, but merely intended as a means for demonstrating various approaches to concurrent object-oriented programming. The discussion is structured around the applied techniques for scheduling received requests, also denoted by the term *synchronisation scheme* [Matsuoka 90]. In [Papathomas 91] a related classification scheme can be found.

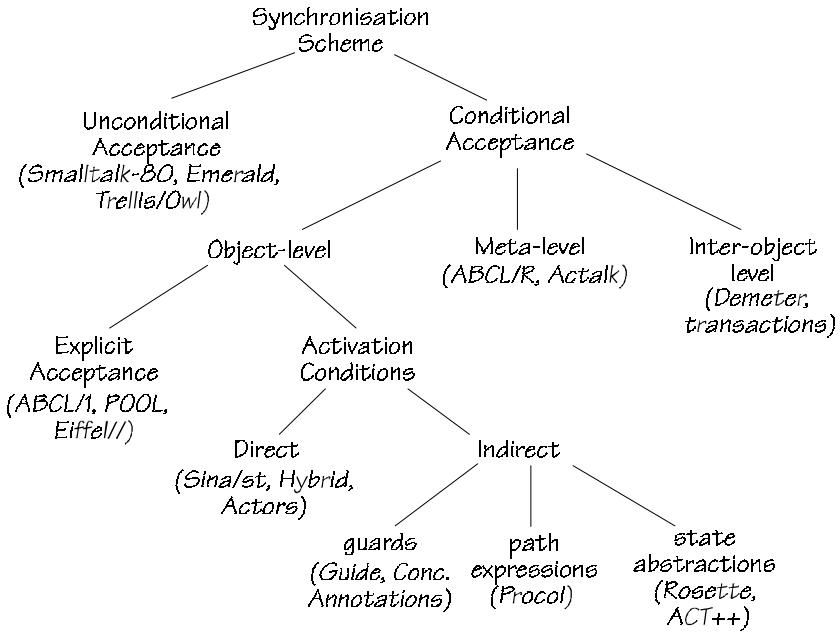


Figure 3.1.7 A classification of synchronisation schemes

Before discussing the various categories in the scheme, we should make explicit that the various languages and systems may belong to more than one category in this classification scheme, even though this may not be apparent from the example systems that are shown. For example, in [Matsuoka 93a] a combination of indirect activation conditions based on state abstractions and a meta-level approach is proposed, one of the key issues in [Matsuoka 93b] is the combination of multiple schemes, and in Guide activation conditions are expressed in a combination of the direct object state and the indirect state (through synchronisation counters).

The first distinction that we make is one that we already discussed before, namely the module of synchronisation. We distinguish conditional acceptance (object level synchronisation) from unconditional acceptance (code level synchronisation). We will focus on the conditional acceptance of messages.

An important classification in three essentially different approaches to the specification of message acceptance synchronisation by objects can be made:

- *object-level specification*: This is the most important category, fitting nicely into the conventional object-oriented paradigm. Every object is responsible for defining its own

synchronisation. Obviously, the specification of this is a part of the class description of the object. In the subsequent text we will focus on this approach.

- *inter-object level specification*: Synchronisation deals with the interdependencies of the various activities in a system. Since these activities propagate through the system by message invocations, imposing (synchronisation) constraints on object-interactions can tackle most synchronisation problems. Examples of this approach are the application of transaction mechanisms (e.g. [Moss 85], [Aksit 91] and [Wakita 91]), Abstract Communication Types (section 2.4.5 of this thesis and [Aksit 94a]) and *synchronisation patterns* [Lopes 94].
- *meta-level specification*: An extension to the object-oriented paradigm are the reflective architectures. For the specification of synchronisation this means that the reception and acceptance of messages can be observed and influenced on a meta-level. Because at the meta-level, the full expression-power of the language is available, virtually any synchronisation scheme can be programmed. Examples are ABCL/R [Watanabe 88], Actalk [Briot 89] and MAUD [Frølund 93].

  Reflective architectures can realise both object-level specification (through *individual-based* reflection) and inter-object level specifications (through *group-wide* reflection), or a combination of these (through a *hybrid* architecture [Matsuoka 91]). But the application of reflection to synchronisation seems to be more pragmatic in approaches that offer a fixed (object-level) synchronisation scheme with a meta-level representation. This is proposed in [Matsuoka 93a] and applied -albeit implicitly- in Rosette [Tomlinson 89a] and Maude [Meseguer 93a, 93b].

As was already indicated, the succeeding text will focus on the object-level approach. This fits in with the object-oriented paradigm, by making objects responsible for their own synchronisation constraints. It brings the associated advantages of encapsulation of implementation, reuse through inheritance, and improved extensibility through modularity.

In contrast, the inter-object level synchronisation techniques deal with synchronisation at another level of abstraction, separating synchronisation from the object specifications. Although this is effective, it conflicts with the autonomy of objects. It should be pointed out, though, that transactions -although these can only partially solve all synchronisation problems- and synchronisation patterns[7] do not cause such problems.

The reflective approach is less attractive for our purposes: a full reflective approach to synchronisation requires a paradigm shift with respect to meta-level programming. Synchronisation code at the meta-level is likely to consist of message expressions implementing synchronisation, instead of a synchronisation specification.

The object level acceptance of messages appears in two forms: through *explicit acceptance* and through *activation conditions*[8]. Explicit acceptance is featured by e.g. ABCL/1, POOL and Eiffel//. It realises synchronisation with code that explicitly considers the state of the

---

7 The reason is that these are converted to object-level specifications in a pre-compilation phase.

8 In [Kafura 89] a similar distinction is made. Here explicit acceptance is termed *centralised control* and activation conditions are termed *decentralised -interface- control* .

object and then determines what message to accept next. This approach, although straightforward, is in general not suitable for synchronisation reuse (these aspects are discussed in detail in the next section).

Two types of activation conditions are distinguished: *direct activation conditions* and *indirect activation conditions*. Approaches that adopt direct activation conditions define explicitly within the method bodies which messages are enabled and which are disabled. This differs from explicit acceptance in that the decision of acceptance is distributed over the relevant methods. Examples of this approach are Actor languages [Agha 86], that can define the subsequent behaviour (including the messages it accepts) of an object with a *become* statement. Other examples are the languages Hybrid [Nierstrasz 87] and Sina/st [Tripathi 88], which employ explicit respectively implicit message queues and can enable or disable the acceptance of messages.

The mechanisms that support indirect activation conditions decouple the acceptance of messages from the implementation in method bodies. Of the many proposed mechanisms, figure 3.1.7 shows three important schemes: *guards*, *path expressions* and *state abstractions*. Guards specify synchronisation constraints for each individual message. Guards are used for instance in Guide [Decouchant 91] and Concurrency Annotations [Löhr 92]. A path expression [Campbell 74] is a specification that defines sequences of allowed requests. Path expressions can only express a limited set of constraints. Extended path expressions [Andrews 83] provide additional guards that can be inserted in a path expression, this can conveniently express most synchronisation constraints. State abstractions describe the state an object is in. After the execution of each method[9], the new abstract state of the object is determined. A separate mapping specifies the messages that are accepted for each abstract state. In ACT++ [Kafura 89], Rosette [Tomlinson 89a] and Synchronizing Actions [Neusius 91a] state abstractions are applied.

The object-level synchronisation schemes will appear again in the next section, where their respective characteristics with respect to reuse and extension of synchronisation code is analysed.

3.1.5 Criteria for COOPLs

Before proposing a list of criteria that we feel a successful concurrent object-oriented programming language (COOPL) should address, we take a look at some criteria, guidelines and requirements that were proposed in other publications.

In [Nierstrasz 91] the following requirements for reusability of COOPLs are defined:

- *Homogeneous model*: all objects must be potentially active, allowing them to protect their internal state.
- *Intra-object concurrency*: objects should allow internal concurrency and support hierarchical composition of active objects.
- *Reply scheduling transparency*: The client object should be able to continue with other activities while waiting for a reply, in such a way that this is transparent to the server object (not necessarily to the client). An example is the use of future RPCs.

9 This technique has been applied mainly in systems that do not support intra-object concurrency.

- *Request scheduling transparency*: The server must be able to block received requests when the object is not ready to serve these (yet). This must be transparent to the client.
- *Incremental modification*: It must be possible to extend objects with new features in subclasses without extensive redefinitions. The cases where this is not possible should be resolvable through reorganisation of the class hierarchy.

These requirements for reuse should allow for the composition of an object from a number of existing objects. Certain aspects of objects, such as application code and synchronisation may be composed from separate objects.

Bloom discusses the evaluation of synchronisation mechanisms in [Bloom 79], independent of an object-oriented context, but very well applicable to it. The discussion concentrates on two main aspects: modularity and the expression of synchronisation constraints:

The modularity requirements are that (a) a module must encapsulate both its synchronisation and the resource, while (b) the synchronisation and the resource must be kept separate. Bloom distinguishes two types of synchronisation constraints: *exclusion constraints* and *priority constraints*. Exclusion constraints ensure the consistency of the encapsulated resources, whereas priority constraints specify the precedence of certain requests over others, usually for efficiency reasons.

Bloom defines the following categories of information that can be used in synchronisation constraints of both types:

- *Type* of the requests (in an object oriented context this would be message selector of the received message).
- *Request reception time*, especially to determine the order in which requests have been received.
- The *parameters* of the request (i.e. message arguments).
- The *synchronisation state* of the resource, which means the state information that is specific to synchronisation.
- The *local state* of the resource, which in the object oriented model corresponds to the instance variables of an object.
- *History information*; this is information about previously fulfilled requests. For example, this could be the number of requests of a certain type, or the type of the last served request.

For testing these criteria, Bloom evaluates two aspects, being (a) the expressive power of a mechanism and (b) the ease of use. The latter is expressed by the suitability of a mechanism for decomposition and composition of synchronisation constraints. As a test case the readers/writers problem is suggested in three variations: (a) with reader priority, (b) with writer priority and (c) with equal priority.

Grass and Campbell emphasise three forms of modularity [Grass 86]: (a) *resource modularity*, which basically means that each resource should be encapsulated by an abstract data type or object that defines both the operations that are allowed on the resource and the synchronisation of requests. (b) *encapsulation of concurrency*, which means that concurrent activities within a module should be allowed, and (c) *synchronisation modularity*: the

specification of synchronisation constraints should be separated from the resource data abstraction (application code).

Our Criteria

In selecting the criteria for COOPLs we focus on software engineering properties to achieve a language that is suitable for the convenient and effective construction of extensible and reusable concurrent applications. Most of the properties that we require have been discussed in this section.

- *Modular concurrency*: the modularity criterion applies to a number of aspects:
  1. Each entity in the system is a potential module for concurrency, and therefore a system should consist of active objects only.
  2. Each object may encapsulate multiple threads, and may encapsulate objects that contain one or more threads as well: intra-object concurrency must be supported.
  3. Synchronisation specifications and application code must be separated.
  4. Synchronisation must be decomposable into modules that can be selected and composed separately[10].
- *Expression-power*: Synchronisation constructs of a language must be general and powerful. A language should provide a simple synchronisation scheme on a per object basis that can be conveniently applied to build *tailored* concurrency and synchronisation constructs. Preferably, synchronisation can be based on a wide range of information types, for example as defined in [Bloom 79].
- *Encapsulation*: the introduction of concurrency and synchronisation should not cause breakage of encapsulation, in particular in combination with inheritance for reusing and extending concurrent objects. For example the synchronisation policy of an object is an implementation aspect that should not be affected by extensions in a subclass. Another example is the protection of local data (instance variables) against concurrent access: this should not be violated by subclasses.
- *Support reuse and composition of concurrent objects*: We want to be able to reuse and extend concurrent objects (i.e. objects with synchronisation specifications). We also want to be able to independently extend or reuse (following synchronisation modularity) both the application code and the synchronisation code. Extension and composition of concurrent objects should be possible without requiring extensive redefinitions.
- *Efficiency*: The mechanisms introduced by a language must be suitable for efficient implementations, the synchronisation mechanism should not incur too much inherent overhead.

These criteria are sometimes overlapping with each other. For example, encapsulation and modularity are closely related, and they both improve the reusability properties of an object. In addition to these criteria, there are some favourable properties in a concurrent language that we want to address as well:

[10] In section 3.2 it will appear that this is an essential property for successful integration of synchronisation code within the object-oriented model.

- *Declarative synchronisation specifications*: a specification of the synchronisation constraints at the interface has a number of advantages; it can offer a clearer, more high-level description of the behaviour of an object. It may support reasoning[11] about specifications, which can be useful for both the software developer and for obtaining efficient implementations.
- *Data-consistency*: A concurrent system designer has to struggle between two conflicting goals: to maintain the consistency of the information stored in the system, while providing maximum accessibility to information, including the ability to refer to, delete or update the data. Concurrent executions can cause inconsistencies if one activity refers to data while another activity is modifying it. Therefore, concurrent programming languages should provide tools such as mutual exclusion for maintaining data consistency.
- *Model Integration*: The features for concurrency and synchronisation should be smoothly integrated within the computation model of the language. In particular, we would like the features for reuse and extension to apply to concurrent objects as well.

We will use these criteria later to judge the concurrent composition-filters object model that is presented later in this chapter.

3.1.6 About this Chapter

In this section several approaches to the integration of concurrency with the object-oriented model have been described. It is generally agreed that objects blend well with concurrency. However, concurrent programs are more complex than sequential ones. The properties of the object-oriented paradigm can help in managing complexity however. Encapsulation can hide and protect resources, inheritance can be applied to reuse concurrent objects.

Thus, not only fits concurrency well into the object-oriented model, concurrent systems can benefit from the object-oriented paradigm in return. We are in particular interested in the reuse and extension of concurrent objects. This can be applied for the construction of libraries and frameworks for synchronisation techniques (see e.g. [Maekawa 80], [Johnson 88a], [Campbell 93])

This chapter is organised as follows: in the next section we discuss the problems of reusing and extending concurrent objects. A framework for synchronisation schemes is defined, and the reuse problems are explained using the framework. This leads to a number of requirements on languages for avoiding the reuse problems. In section 3.3 the composition-filters approach to concurrency and synchronisation is introduced. First the means for creating concurrent threads are described, then the synchronisation of messages on the interface of objects is described. In section 3.4 these techniques are applied to a number of example problems. Section 3.5 presents an evaluation and related work.

---

11 The reason is that a declarative synchronisation specification may offer a more restricted model with well-defined semantics. This makes it much easier to reason about it, compared to synchronisation statements that are embedded within a fully expressive programming language.

3.2 Inheritance Anomalies in Concurrent Programming

This section discusses the so-called *inheritance anomalies*; problems that arise with the reuse of synchronisation code. The concept of inheritance anomaly is explained, and a generic framework for synchronisation schemes is introduced. Then the origins of the inheritance anomaly are discussed, divided into three categories. For each category the anomalies are illustrated with concrete examples and explained using the framework for synchronisation schemes. The section concludes with a summary and a discussion of closely related work.

3.2.1 Reuse and Extension of Concurrent Objects

In section 3.1, we discussed the approaches for synchronisation in an object-oriented model. The most important approach is that of *interface control*, which means that the object synchronises messages upon arrival. The synchronisation constraints of an object are defined as a part of its specification and implementation. Two goals of object-oriented concurrent programming are (a) to support effective concurrent programming through reuse, and (b) to improve the modelling power through inheritance hierarchies.

Concurrent programming is generally considered to be more difficult and error-prone than conventional, sequential programming. The object-oriented paradigm supports effective software development through encapsulation, polymorphism and inheritance. In particular, inheritance and related reuse mechanisms can be applied to reuse pre-defined, well-tested code. Reuse of stable concurrent code may have even more impact on the development effort for concurrent software.

The classification relations that are identified in object-oriented analysis result in hierarchies of objects. Classes in this hierarchy inherit properties from their parents (both direct and indirect), and extend these with new properties. This applies to methods and instance variables, but to synchronisation properties as well.

However, the reuse and extension of concurrent objects appears to be far from trivial: several researchers have indicated (e.g. in [America 87], [Briot 87], [Kafura 89] and [Nierstrasz 91]) that there is an interference between inheritance and concurrency. In particular, conflicts occur due to the inheritance of synchronisation specifications in subclasses. These conflicts typically reveal themselves by enforcing superfluous redefinitions in a subclass. For example, introducing a new method or overriding an inherited method in a subclass may require additional redefinitions of seemingly unrelated methods. In [Matsuoka 90], the term *inheritance anomaly* was coined to designate such problems.

These problems have even caused language designers to abstain from adopting inheritance as a primary data abstraction technique (e.g. POOL-T [America 87] and ABCL/1 [Yonezawa 90]). In a number of other languages, classes with synchronisation cannot be extended (e.g. Dragoon [Atkinson 91]), or only if the synchronisation is fully re-specified (e.g. Eiffel// [Caromel 90] and POOL-I [America 90]).

It is important to note, however, that this problem is not inherent to the combination of concurrency and the object-oriented paradigm. Instead, the inheritance anomaly is

completely due to the particular synchronisation scheme and inheritance semantics of a language. In the case that these are too restricted to express certain desired behaviour, the programmer must 'program around' the problem in method implementations. It turns out that this often requires the redefinition of seemingly unrelated methods.

In this section, we will describe and analyse the inheritance anomalies and define a set of criteria for avoiding them. We would like to stress that the identification and description of inheritance anomalies was first made by Matsuoka, Yonezawa and Wakita in [Matsuoka 90] and [Matsuoka 93a]. However, the analysis presented here is different in the sense that it focuses more on the origins rather than a classification of the anomalies. A first attempt towards the analysis presented here was made in [Bergmans 92a].

The analysis starts with a description of a framework for synchronisation schemes. Subsections 3.2.4 to 3.2.6 then proceed with a discussion of three categories of sources for the inheritance anomaly. Section 3.2.7 gives some conclusions.

3.2.2 A Generic Framework for Synchronisation Schemes

In this subsection we make an attempt at describing the fundamentals of synchronisation in an object-oriented concurrent computation model. By constructing a precise model of the synchronisation schemes, we are able to explain the potential problems in inheriting concurrent code, and we can analyse why previously proposed mechanisms fail in overcoming the inheritance anomaly.

We assume the following simplified object model:

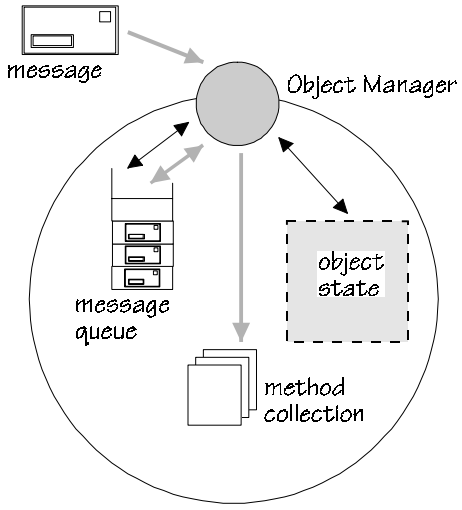


Figure 3.2.1 A simplified model of the synchronisation of messages

In this model, each object has an *object manager*, a message queue, a collection of methods and an object state. The *object manager* is responsible for scheduling messages: based on the state of the object, the state of the message queue and the synchronisation constraints, it is decided whether a received message can result in immediate method execution, or whether it will be put in the message queue. The object manager also takes care that queued messages will eventually lead to the execution of a method, once their synchronisation constraints are satisfied. The object manager operates on one message at a time to ensure that competing messages are handled consistently.

If we consider the synchronisation with more detail, we can identify a number of factors that determine whether a message is acceptable or not: as stated before, message acceptance ultimately depends on the state of object, in the broadest sense possible. We term this the *abstract object state*, to avoid confusion with the common use of object state to designate the values of instance variables only. The abstract object state may include the values of instance variables, the properties of the message, such as the message selector and the message arguments, the current activities within the object (for example, the number of active threads), the message queue (i.e. what other messages are waiting to be served), and the history of the object (for instance, which messages have been executed and how often). Depending on the particular object model, properties may be added to or removed from this list.

The abstract object state is affected by several types of events: the arrival of new messages, the acceptance of a message or start of a message execution, the termination of a method execution, and the effects of method executions themselves. The dependencies are outlined in the following diagram:

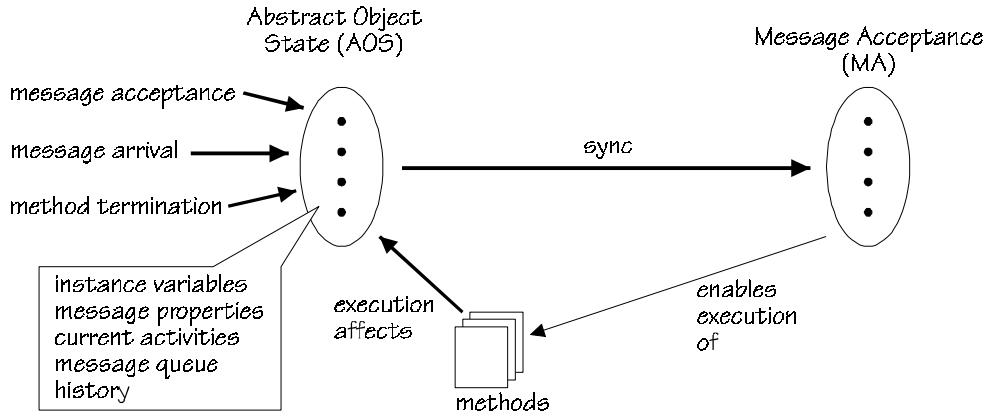


Figure 3.2.2 Schematic outline of (one-step) synchronisation specification

In this figure the abstract object state, *AOS*, is modelled by a set consisting of elements that each represent one particular state of the object. Synchronisation of messages is modelled by a set of accepted messages (so-called *accept set*). Thus, synchronisation is the mapping between the abstract object state *AOS* and the set of possible accept sets of an object, *MA*. this is formalised as follows:

Definition 3.2.1: One-step synchronisation specification

Assume an object *o* with a -current- abstract object state $s \in AOS$ and a -current- accept set $a \in A$.

(1) A message *m* is accepted by *o* if and only if $m \in a$.

(2) The synchronisation specification of an object, *sync*, is the mapping from *AOS* to *MA*:
sync : $AOS \rightarrow MA$, and thus $sync(s) = a$

As a refinement of this model, we introduce an intermediate between the object state and the message accept set, which we term *synchronisation conditions*. The synchronisation conditions can be considered intuitively as an abstraction, or characterisation, of the abstract state of the object. The abstract object state is expressed as a set of satisfied conditions.

This is formally represented as a set of identifiers, where each identifier represents a satisfied condition. The definition of synchronisation then becomes:

Definition 3.2.2: Two-step synchronisation specification

Assume an object *o* with an abstract object state $s \in AOS$, synchronisation conditions $c \in SC$ and an accept set $a \in MA$.

(1) A message *m* is accepted by *o* if and only if $m \in a$.

(2) The *state abstraction sa* represents the mapping between the object state space *AOS* and the synchronisation conditions *set SC*; $sa: AOS \rightarrow SC$.

(3) The *condition mapping cm* represents the mapping between the synchronisation condition set *SC* and the message acceptance space *MA*; $cm: SC \rightarrow MA$.

(4) The synchronisation specification of an object, *sync*, is the mapping from *AOS* to *MA*: $sync : AOS \rightarrow MA \stackrel{def}{=} sa \circ cm$, and thus $sync(s) = cm(c) = a$, where $c = sa(s)$

The reasons for introducing this staged synchronisation scheme are two-fold: firstly, it is necessary to properly model the synchronisation specifications in a number of COOPLs. Secondly, it will be shown that such a staged model is necessary for effectively reusable and extensible active objects. The following figure graphically depicts this staged synchronisation scheme:

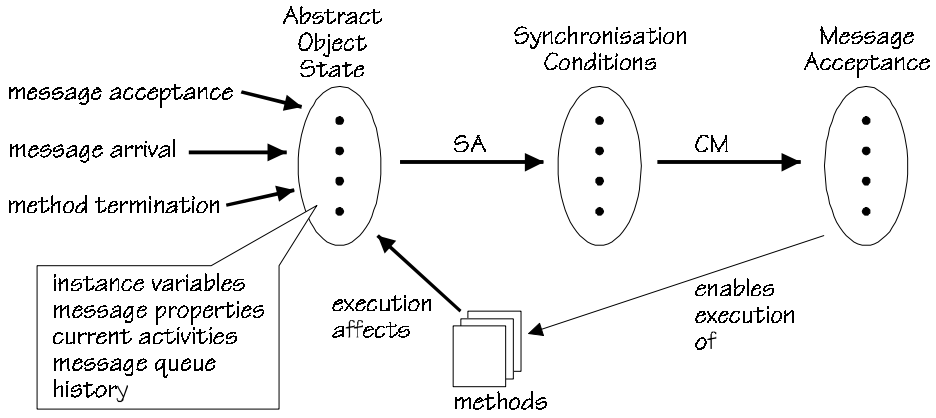


Figure 3.2.3 Schematic outline of two-step synchronisation specifications

To demonstrate how this model corresponds to actual synchronisation schemes, consider the language ACT++ [Kafura 89]. This language specifies synchronisation through so-called *behavioural abstraction*. For this purpose, every object defines a set of behaviour names and for every behaviour name the methods that are accepted for that behaviour. Every method specifies the next behaviour with a *become* statement. The behavioural abstractions in ACT++ correspond to the synchronisation conditions in our formal model.

The become statements implement the mapping from the abstract object state to the synchronisation conditions, and the definition of acceptable methods for each behaviour realises the mapping between synchronisation conditions and the message accept set.

In some synchronisation schemes, the mapping from the abstract object state to method acceptance is made immediately, without the intermediate step of synchronisation

conditions[1]. This is the case in guard-like approaches such as Guide [Decouchant 91]; a method guard directly maps the current state of the object onto message acceptance. We will use this framework for synchronisation schemes to describe the problems with reuse of synchronisation code in a language-independent way, and we claim that most COOPLs that support interface control and object-level synchronisation can be mapped onto this model.

3.2.3 The Origins of Inheritance Anomaly

To illustrate the conflicts between inheritance and synchronisation, we use the well-known bounded buffer example[2]. This is an example of a shared resource that must be protected against simultaneous access, and that requires synchronisation between different clients. A bounded buffer object provides a get and a put operation, which respectively remove from and add elements to the buffer.

In two boundary situations this requires synchronisation (except for mutual exclusion, that we assume in all cases). The first case is when the buffer stores no element at all: in this case a get operation must be blocked until new elements are added (through put operations). The second case is when the buffer is filled to its limit with elements; then no more put operations should be accepted until additional space comes available.

We will use pseudo-code for describing the examples. The pseudo-code is based on the syntax of Sina, but will be extended when necessary with constructs that are specific to the particular synchronisation schemes. The definition of the bounded buffer without synchronisation is as follows (we omit the implementation details of the methods):

```
class BoundedBuffer
    inherits Object  // declare superclass(es)
    constants
        limit = 10;  // the maximum number of elements in the buffer
    instvars  // instance variable declarations
        store : Array(limit);    // index ranges from 1 to <limit>
        head, tail : Integer;
    methods  // the methods on the interface of the object.
        get returns Any;   // returns the element pointed at by <tail>
        put(newElem:Integer) returns Nil;   // stores the argument at position <head> in
        <store>
end;
```

In the remainder of this section several extensions to the bounded buffer will be introduced to illustrate the various problems that can be encountered when trying to extend concurrent objects. The origins of these conflicts between inheritance and synchronisation can be divided into the following three categories:

1. Synchronisation modularity.
2. Synchronisation granularity.
3. Expressiveness for synchronisation conditions.

---

1 Although this can be modelled as having a one-to-one mapping between synchronisation conditions and acceptable messages.

2 Although this is not a very creative choice, the bounded buffer and its variations have become a canonical example for demonstrating expressiveness and reusability of synchronisation schemes.

In the next three subsections these three categories are discussed one by one.

3.2.4 Synchronisation Modularity

The modularity of synchronisation specifications is an important requirement for effective reuse and extension. It means that the synchronisation specification is separated from the application code of the object. The importance of synchronisation modularity has been widely recognised (yet widely ignored in COOPLs as well), e.g. in [Bloom 79], [Grass 86] and [Frølund 92][3].

The prime reason for requiring synchronisation modularity is to reduce the dependencies between the application code, such as method implementations, and the acceptance of messages. Because in subclasses the synchronisation will virtually always change (if only to cope with newly added methods), dependencies between application code and message acceptance will require the redefinition of application code in many situations.

We discuss three levels of synchronisation modularity: the synchronisation in so-called *bodies*, explicit message acceptance and specification of synchronisation conditions in methods.

Synchronisation in Bodies

In a number of COOPLs each object defines a part called the *body*; this is code that is executed independently from the execution of incoming messages (e.g. POOL-I [America 90], Eiffel// [Caromel 90]). In these systems, the body represents the high-level specification of a process that is associated with the object: the received messages are considered and when deemed appropriate (i.e. acceptable) the execution of corresponding methods is initiated (i.e. the process performs them).

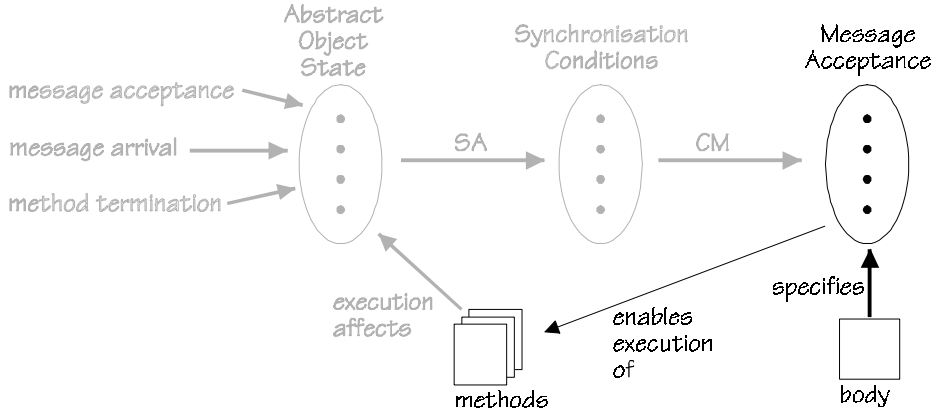


Figure 3.2.4 Schematic representation of body synchronisation.

The previous figure visualises the dependencies. The parts and relations that play no role are greyed out. The body can be considered to be a special type of method. The possibility of

[3] We should mention the approach taken by Meseguer [Meseguer 93a, 93b], to completely eliminate synchronisation code by replacing the conventional method implementations in terms of message expressions by so-called *rewrite rules*. This is based on the same motivation, but taken to an extreme.

inheritance anomaly can be detected by the presence of a direct connection between the body and message acceptance.

Except for realising the synchronisation constraints of the object, the body may implement some general house keeping operations as well. The problem with this approach appears when additional synchronisation semantics are required in subclasses; in this case the whole body must be defined again. Most important, in the subclass all the synchronisation constraints for the methods defined in parent classes must be programmed again. This conflicts with incremental specification. It also violates encapsulation, because the synchronisation constraints that are implementation-dependent must be redefined as well. This requires knowledge of the implementation of the superclasses, and makes the subclass dependent of the implementation of the superclasses.

Explicit Specification of Message Acceptance

A closely related approach is through explicit message reception. Some languages (e.g. Ada, ABCL/1) provide the programmer with facilities to put explicit message reception statements in the application code. The drawbacks of explicit message reception are two-fold: firstly, when the programmer extends an existing synchronisation constraint through subclassing, the complete method body must be redefined. Secondly, each individual method becomes responsible for defining message acceptance. Thus the extensions in subclasses must -in the worst case- be taken care of by all methods, requiring their redefinition.

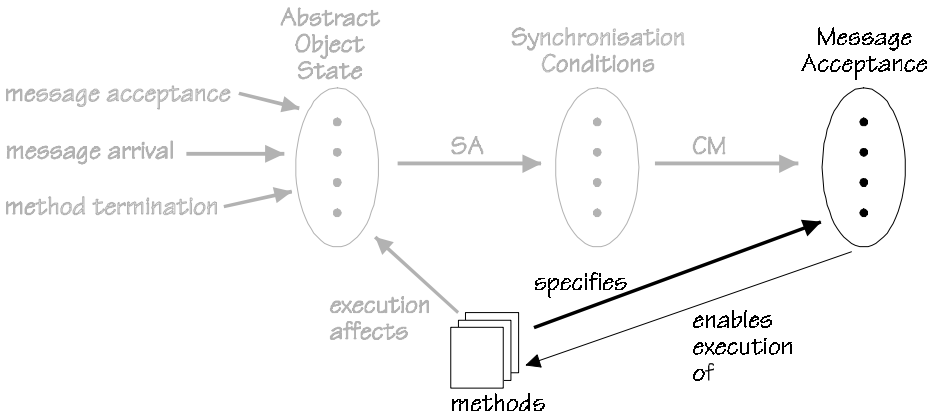


Figure 3.2.5 Schematic representation of explicit message acceptance.

The figure illustrates the essence of the problem: the tight coupling between the method implementations and message acceptance prohibits the adaptation to changes and extensions in subclasses.

Specification of Synchronisation Conditions in Methods

To overcome the problems that are encountered by the previously discussed approaches, the concept of behavioural abstraction has been introduced in [Kafura 89]. Behavioural abstractions are a form of synchronisation conditions: they represent an abstraction of the current state of the object, with an associated set of acceptable messages. At the end of each method a *become* statement specifies the new state the object will be in. In pseudo-code, the definition of a bounded buffer object with behavioural abstraction may look as follows:

```
class BoundedBuffer
    inherits Object  // declare superclass(es)
    constants
        limit = 10;  // the maximum number of elements in the buffer
    instvars  // instance variable declarations
        store : Array(limit); // index ranges from 1 to limit
        head, tail : Integer;
    behaviour   // here the behavioural abstraction with associated accept sets are defined
        empty = { put };
        full = { get };
        partial = { put, get }
    methods  // the methods on the interface of the object.
        get returns Any;
            begin ... if head=tail then become empty  else become partial ;  end;
        put(newElem:Integer) returns Nil;
            begin ... if tail-1=head.mod(limit) then become full  else become partial ;  end;
end;
```

Some important properties of this approach are that (a) in each method the new state after the method execution must be determined explicitly, (b) the object can be in only one -abstract- state at a time, and (c) with each abstract state the set of associated methods is hard-coded.

Now consider a subclass labelled TestBuffer, introducing a new method, isEmpty, that tests whether the buffer is empty. As this method has no side-effects and can be executed in any state of the object, it would be natural for the definition of this method to be independent of any synchronisation specifications. Unfortunately, this is not the case, as is shown by the following definition of TestBuffer:

```
class TestBuffer
    inherits BoundedBuffer
    behaviour
        empty = { put, isEmpty };
        full = {get, isEmpty };
        partial = { put, get, isEmpty }
    methods  // the methods on the interface of the object.
        isEmpty returns Boolean;
            begin
                ...
                if tail-1=head.mod(limit) then become full
                elseif  head=tail then become empty
                else become partial ;
                ...   // and return the result.
            end;
end;
```

The introduction of the (synchronisation-less) isEmpty method requires redefinitions related to synchronisation at 2 points: firstly, all the behavioural abstractions must be redefined to incorporate the new method. Apart from being tedious, this is an important shortcoming: when the interface of a superclass changes, in all the subclasses the behavioural abstractions must be redefined. This happens if methods or behavioural abstractions are added, removed or renamed.

The second location in the definition of TestBuffer where synchronisation must be coped with explicitly is at the end of the isEmpty method: here a *become* is required to define the

new behavioural abstraction of the object, even though its state has not changed. This requires a complete analysis of the current state of the object with the corresponding *become* statements. The problem is that the synchronisation, even for inherited methods, is fully implemented here again. Thus, a change to the synchronisation in class BoundedBuffer may require updating the methods in all its subclasses.

For brevity, in our examples we directly access the instance variables of the superclasses. This in fact breaks encapsulation, as the internal data structure of a class should be invisible to its subclasses [Snyder 86]. In our examples, this can be easily overcome by adding methods for accessing the relevant instance variables in the (super-)class, and replacing access to these instance variables by calls to corresponding methods. This does *not* affect the point that the examples are making; the unnecessary redefinitions of synchronisation specifications. In the case of the classes BoundedBuffer and TestBuffer, methods should be added to BoundedBuffer for retrieving the values of the head and tail instance variables, respectively. It would also be possible to put more abstract methods on the interface of BoundedBuffer, such as for retrieving the current size of the buffer, or even methods for testing whether the buffer is empty or full. This would still require the redefinitions that we pointed out.

The concept of behavioural abstractions is visualised in the succeeding figure:

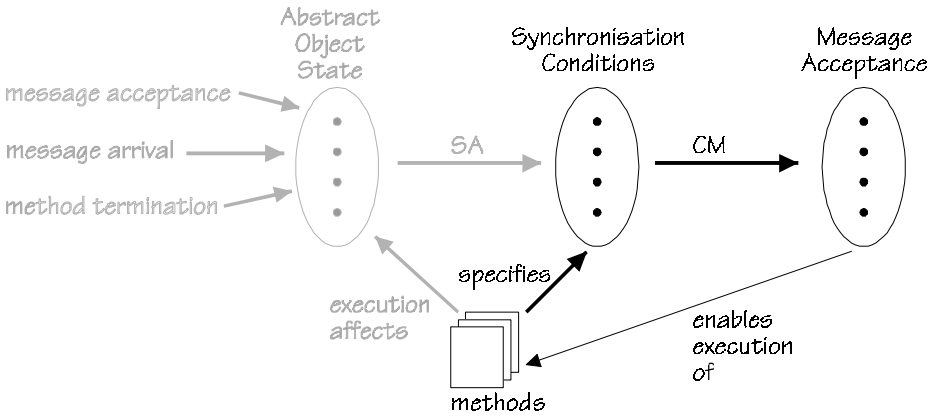


Figure 3.2.6 Schematic representation of specification of synchronisation conditions.

One of the problems is that the coupling between the method implementations and the synchronisation conditions (behaviour abstractions in this case) is still too tight: each method explicitly specifies the next synchronisation condition. This problem can be partially solved by specifying the state analysis and *become* statements separately, and sharing them among multiple methods. An example of this are the default *transition specifications* in ABCL/onAP1000 [Matsuoka 93b]. This makes it unnecessary to define the same become statements repeatedly.

The problem of the redefinition of the behavioural abstractions is due to the strict one-to-one mapping between synchronisation conditions and message accept sets (the relation *CM*). For instance in Rosette [Tomlinson 89a] this problem is addressed by making the behavioural abstractions first-class objects called *enabled sets*. A number of operations are defined on enabled sets allowing the definition of an enabled set as a combination of other enabled sets:

```
class TestBuffer' // demonstrating enabled set approach
    inherits BoundedBuffer
```

```
behaviour
    empty = {isEmpty} + super(empty);  // extends the empty set defined by the
    superclass
    full = {isEmpty } + super(full);
    partial = {isEmpty } + super(partial);
... // the rest is similar to the previous definition of TestBuffer
```

Although this definition is not shorter than the previous version of TestBuffer, it can cope with changes to the superclasses[4]. More complex enabled set expressions are possible, but these will be discussed in the context of the granularity of synchronisation specifications in subsection 3.2.5.

One problem with the specification techniques that we have seen until now can be observed in figure 3.2.6: when due to the lack of synchronisation modularity the execution of a method, directly or indirectly, determines synchronisation, this requires a *prediction* of the actual object state at the moment a message arrives. The state of the object, however, can change in the meanwhile, due to the reception or acceptance of other messages. The techniques based on *become* statements at the end of a method, although intricate and powerful, cannot cope with such problems. Related to this, intra-object concurrency cannot be dealt with properly because the analysis prior to a *become* can be invalidated immediately afterwards by another active thread.

Consider for example an extension to the BoundedBuffer class that introduces a method lowPriorGet. This method has a lower priority than the normal get method, meaning that no lowPriorGet message will be accepted if there is a get message in the queue. Assuming that we have means to test the current number of blocked get messages in the queue, determining message acceptance within methods can not be specified correctly.

3.2.5 Synchronisation Granularity

The granularity of synchronisation specifications is an important factor for effective reuse and extension of synchronisation specifications. This is easily understood when considering a synchronisation specification that is a single, monolithic unit. An example of this are *path expressions* [Campbell 74]. A path expression is a single expression that imposes constraints or a relative ordering upon requests. Path expressions extended with guards can conveniently express most synchronisation constraints [Andrews 83].

Specialisation, extension or reuse of such monolithic synchronisation specifications in subclasses is not possible in the general case. It is not possible to replace parts of the specification. Adding new constraints to the specification is only feasible in specific cases (when the additional constraints are fully orthogonal to the existing specification). We argue that this is due to the large granularity of the specification, which does not allow the selection or replacement of a part of the synchronisation specification.

Consider for example a path expression that controls access to a resource such as a file (based on an example in [Laffra 92]). The ";" symbol defines sequence, the "+" denotes a selection and "*" designates repetition:

```
(OpenForRead ; ReadRecord* ; Close) + (OpenForWrite ; WriteRecord* ; Close)
```

---

4 Inheritance anomalies are not about saving keystrokes...

This expression allows repeated read or write actions only when preceded by a corresponding open statement. Now suppose that we want to add (in a subclass) a new operation ReadCharacter. This has the same constraints as hold for the ReadRecord operation. A path expression that satisfies this is as follows:

```
(OpenForRead ; (ReadRecord + ReadCharacter)* ; Close) +
 (OpenForWrite ; WriteRecord* ; Close)
```

However, the only way to achieve this is by completely redefining the existing path expression. There is no straightforward way to extend the path expression with new constraints, or to reuse the specification of constraints for ReadRecord. The illustrated problems appear in languages that adopt monolithic synchronisation specifications, such as PROCOL [Bos 89], that adopts extended path expressions, POOL [America 87, 90] and Eiffel// [Caromel 90].

This example illustrates two important aspects of synchronisation granularity: the composition of a synchronisation specification from two or more components (this covers extension as well), and the reuse of a constraint specification in a different context (e.g. another class, or other methods). The latter is concerned with the polymorphic application of a (part of a) synchronisation specification to other objects or methods. These two issues will next be discussed separately.

If we consider the granularity problem in the schematic representation of synchronisation mechanisms, we must focus on the synchronisation conditions and on the relations *SA* and *CM*. The synchronisation conditions provide for an intermediate that is independent of both the implementation details of the object and the precise interface specification. This makes it suitable as a reusable abstract synchronisation specification without violating encapsulation and that can be tailored to the exact interface of the reusing client class:

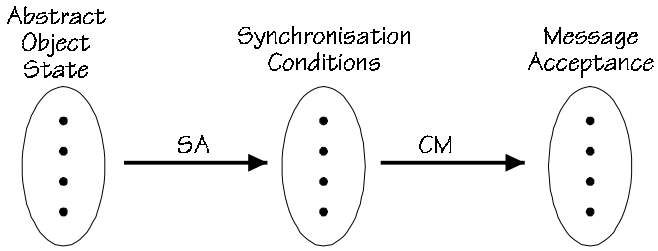


Figure 3.2.7 The synchronisation conditions introduce a level of granularity.

Note that synchronisation conditions do not imply a particular synchronisation scheme; they can range from behavioural abstractions to guards. The relation *SA* describes the mapping between the abstract object state and the synchronisation conditions, and *CM* describes the relation between the synchronisation conditions and the acceptance of messages.

Polymorphic Synchronisation Specifications

The requirement for polymorphic synchronisation specifications is based on the goal of reusing and extending synchronisation specifications. Consider for example the possible extensions to bounded buffer. We give a few examples:

```
getLatest : returns and removes the element of the buffer that was added latest.
getX(x) : returns and removes the x-th element in the buffer
peekFirst : returns, but does not remove, the first element of the buffer
smallest : returns the smallest element in the buffer
```

For each of these example methods a range of variations can be imagined. The notable thing is that all these methods require the same synchronisation as the simple get method: they can be applied only when there are elements in the buffer, otherwise they cannot return a result. Instead of specifying the same synchronisation code repeatedly, the polymorphic application of the synchronisation constraints of get to the other methods is preferable.

Polymorphic application of synchronisation constraints is particularly important in cases where the synchronisation depends on implementation aspects of the (super-)class, and the constraints are reused in subclasses. A change in the implementation of the superclass may require a redefinition of the synchronisation constraints. This should remain hidden to the subclasses, which does not hold when the synchronisation constraints have been re-implemented.

We show that guard-like approaches such as Guide [Decouchant 91], *Concurrency Annotations* [Löhr 93] and the *synchronisers* in [Matsuoka 93b][5] suffer from these problems. The bounded buffer example is redefined with synchronisation by guards: for each method a guard condition is defined that specifies a necessary condition for the acceptance of the corresponding method:

```
class BoundedBuffer
    inherits Object  // declare superclass(es)
    constants
        limit = 10;  // the maximum number of elements in the buffer
    instvars  // instance variable declarations
        store : Array(limit); // index ranges from 1 to <limit>
        head, tail : Integer;
    guards   // we assume a guard approach
        get : ( head<>tail ) ;
        put : ( tail-1<>head.mod(limit) );
    methods  // the methods on the interface of the object, we omit the implementations.
        get returns Any;
        put(newElem:Integer) returns Nil;
end;
```

We define a subclass that adds two methods, peek and getLatest, with synchronisation constraints that are equal to those of the get method:

```
class PeekBuffer
    inherits BoundedBuffer  // declare superclass(es)
    guards   // we assume a guard approach
        peek : ( head<>tail )  // re-implementation
        getLatest : ( head<>tail )  // re-implementation
    methods  // the methods on the interface of the object, we omit the implementations.
        peek returns Any;  // returns the element at the head of the buffer.
        getLatest returns Any;  // returns and removes the latest added element.
end;
```

The problems with this implementation arise for example when we change the implementation of the BoundedBuffer such that the head and tail instance variables are no longer available. Assume that the new implementation is as follows:

---

5 Although this is less severe because their approach supports behavioural abstractions as well, which can cope with this particular problem.

```
class BoundedBuffer
    inherits Object  // declare superclass(es)
    constants
        limit = 10;  // the maximum number of elements in the buffer
    instvars  // instance variable declarations
        store : Collection(limit); the array is replaced by a collection object of variable size
    guards   // we assume a guard approach
        get : ( store.size>0 ) ;
        put : ( store.size<limit ) ;
    methods // the methods on the interface of the object, we omit the implementations.
        get returns Any;
        put(newElem:Integer) returns Nil;
end;
```

This requires the modification of the subclass PeekBuffer as well. We should remark that, if message invocations are allowed as a part of the guard expressions (e.g. in Guide this is not allowed), the problem can be circumvented. The implementation of the synchronisation constraints must then be done in separate methods of class BoundedBuffer, e.g. a method nonEmpty, and in class PeekBuffer the guard expressions must be replaced with an invocation "server.nonEmpty". The method that implements the guard expression can be applied polymorphically, which solves the problem. Note that this requires a separate method implementation for every guard expression in an object.

In our schematic model of synchronisation mechanisms, the property of polymorphic synchronisation specifications is a characteristic of the mapping between the synchronisation specifications and the message acceptance. In particular, with each synchronisation condition multiple acceptable messages must be associated, and it must be possible to extend the mapping incrementally, in order to associate additional messages with an existing synchronisation condition (as exemplified by the union operation on enabled sets in [Tomlinson 89a]).

Synchronisation Composition

The term composition applies to the composition of multiple predefined components as well as the extension of a predefined component with new, local, components. Synchronisation composition is essential for the reuse and extension of synchronisation constraints. In some systems this objective has been discarded, usually motivated by the complexity of the problem and the relatively small amount of reused code. Examples of this are POOL-I [America 90], Eiffel// [Caromel 90] and Dragoon [Atkinson 91], [Reghizzi 91]. In these systems synchronisation specifications can be inherited (once), but not extended afterwards.

The inheritance and extension of synchronisation is more than a reduction in the amount of code that must by typed, though. The inheritance and incremental modification of classes that incorporate synchronisation is essential for effective component reuse. We already demonstrated that re-implementation of synchronisation constraints in subclass may violate encapsulation of the superclass.

We illustrate the problems in synchronisation composition with an example of multiple inheritance. Consider a class LockingBuffer that inherits from both class BoundedBuffer and class Locking. Class LockingBuffer is a bounded buffer that can be locked and unlocked. If the buffer is locked no methods are accepted for execution, except the method

unlock that removes this restriction. The functionality of locking and unlocking of messages is defined in class Locking.

The -partial- definition of class Locking is shown. In the example synchronisation is expressed using behavioural abstractions:

```
class Locking
    inherits Object  // declare superclass(es)
    behaviour // we adopt the behavioural abstraction approach
        locked = {unlock};
        unlocked = {lock};
    methods // the methods on the interface of the object.
        lock returns Nil; begin ... become locked end;
        unlock returns Nil; begin ... become unlocked end;
end;
```

The problems with the realisation of LockingBuffer are two-fold: firstly, the synchronisation constraints that are defined by the two classes are sometimes contradictory. For example, suppose an instance of LockingBuffer is in the locked state, and contains a few elements. Class Locking dictates that no method except unlock is to be accepted. However, according to class BoundedBuffer, the methods put and get should be acceptable. These conflicting constraints must be resolved, which is not trivial, in particular in the general case. For example, synchronisation that is based on behavioural abstractions is faced with the problem that the abstract state space of the object is multiplied due to the combination of two orthogonal state spaces (i.e. *empty*/*partial*/*full* versus *locked*/*unlocked*).

The second problem is that the synchronisation constraints defined by Locking are to be applied on the methods of a subclass. These methods are not known yet when Locking is defined. It is thus important to define a form of *open-endedness* in the synchronisation specification. This is achieved in [Frølund 92] through a construct called *all-except*, which defines constraints for an open-ended set of methods, including those in future subclasses. Another approach to solving this is by defining operations on message accept sets that allow for extending them in a subclass (e.g. in Rosette [Tomlinson 89a] and in [Matsuoka 93b]). This makes the accept sets first-class objects, a form of reflection. Approaches that are based on method guards cannot do this.

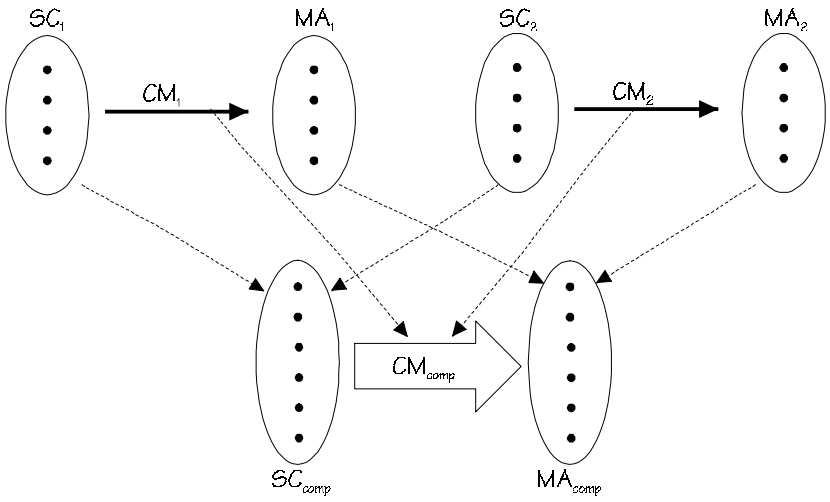


Figure 3.2.8 Illustration of synchronisation composition

We briefly consider how the problems of synchronisation composition appear in the formal model for synchronisation mechanisms: there is a difference with the previous aspects, which dealt with the properties of the synchronisation mechanism itself, whereas now we are interested in the combination of two synchronisation specifications. Assume that we want to compose two synchronisation specifications, $Spec_1$ and $Spec_2$. Each of these specifications consists of a condition mapping CM_i: $SC_i \rightarrow MA_i$ and a state abstraction function SA_i. Because the state abstraction functions SA_i are implementation specific, we are only interested in combining CM_1 with CM_2. The combination of synchronisation specifications is illustrated in figure 3.2.8.

The following two situations can be distinguished when composing synchronisation:

(a) CM_1 and CM_2 are orthogonal: this is the case when they deal with different sets of messages: $(\forall as_1 \in MA_1)\ (\forall as_2 \in MA_2) \bullet as_1 \cap as_2 = \emptyset$. In this case they can be easily combined into a new condition mapping:

$$CM_{comp} : (SC_1 \cup SC_2) \rightarrow (MA_1 \cup MA_2) \stackrel{def}{=} CM_{comp}(sc) = CM_1(sc) \cup CM_2(sc)$$

Thus, in each state of the object all the messages that were accepted by $Spec_1$ or by $Spec_2$ are now accepted in the composed specification.

(b) CM_1 and CM_2 conflict: when these both define synchronisation constraints for the same message, this is likely to be contradictory. Thus, we must define some operation for combining two condition mapping functions into a new one. Obviously, this is an important issue in the semantics of a synchronisation scheme. For instance in [Frølund 92] it is made explicit that the constraints made in a superclass should never be relaxed.

A special case of conflicting specifications occurs when components define *open-ended* constraints, i.e. that apply to the future methods defined in subclasses as well. These must be treated separately in the formal model, for instance as a higher-level function by itself, operating on the condition mapping function.

3.2.6 Expressiveness for Synchronisation Conditions

The expressive power for specifying synchronisation conditions varies widely among synchronisation schemes. At one end of the scale are mechanisms such as path expressions, which can only express acceptance based on very specific information (i.e. the history of executed messages). Extended path expressions provide guards to extend the expressive power. Another specific mechanism for expressing synchronisation is based on synchronisation counters [Robert 77]. The more sophisticated synchronisation schemes (e.g. in [Frølund 92] and [Matsuoka 93b]) allow -almost- the full expressiveness of the programming language for expressing synchronisation conditions.

Insufficient expressiveness for synchronisation conditions can reveal itself in two -closely related- ways: it may completely prohibit the expression of specific synchronisation constraints. But it may well be possible to 'program around' the problem, by writing additional application code (i.e. in methods). The latter situation is a frequent source for inheritance anomaly.

One particular example of this is the so-called *history-sensitiveness* [Matsuoka 90], which is exemplified by an extension to the bounded buffer. Suppose that we want to add a subclass

of BoundedBuffer that provides a method gget. This is an ordinary get operation, except that it should not execute immediately after a put. The essential problem of this is that it requires the expressive power to access the history of the object. This can be provided by the system (cf. the *wait_once* transitions in [Matsuoka 93b]), or the application must do some explicit bookkeeping.

In the gget example, consider that there is no special language support for keeping a history administration, then the only way to know that the latest executed method was a put is by modifying all the methods of the object to maintain a Boolean variable. This is shown in the following code example, based on guards for synchronisation:

```
class HistoryBuffer
    inherits BoundedBuffer  // declare superclass(es)
    instvars
        justPut : Boolean;
    guards
        gget : (justPut.not)  // only allowed when the previously executed method was not a
        put
    methods  // the methods on the interface of the object.
        get returns Any; begin justPut:=false; return super.get end;
        put returns Nil; begin justPut:=true; return super.put end;
        gget returns Any; begin justPut:=false; return super.get end;
end;
```

Note that we can use the implementations of the superclass methods by using the super pseudo-variable, but this does not make the inheritance anomaly less severe: not only must all methods be redefined (including the ones inherited indirectly), but in future subclasses the justPut Boolean must be maintained as well.

A solution to this bookkeeping problem is available in some systems through the definition of generic pre- and post-actions that are executed for all messages that are received by the object (e.g. in Encapsulators [Pascoe 86], or in reflective languages such as ABCL/R [Matsuoka 91] and Maude [Frølund 93]).

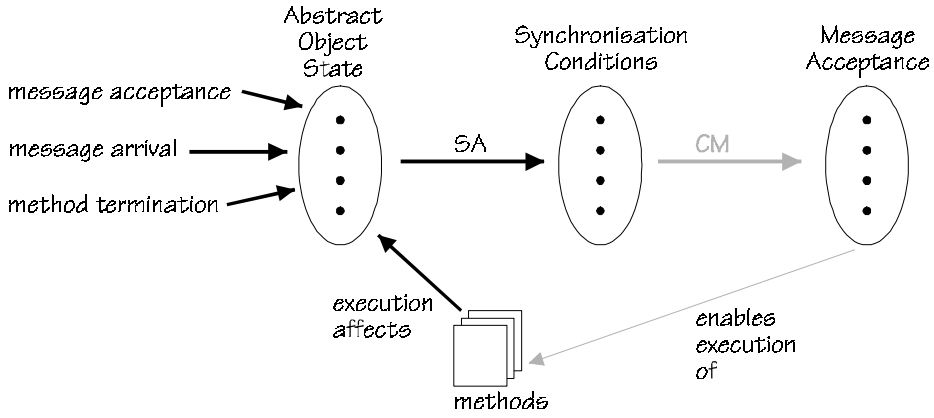


Figure 3.2.9 Schematic illustration of the expressiveness problem

The relevance of this example is to demonstrate how lack of expressiveness can cause inheritance anomalies. Important is the possibility of making the mapping from *any* abstract object state to a synchronisation condition (illustrated by relation *SA* in figure 3.2.9). The problem, however, is that the notion of abstract object state is quite broad and in fact

system-dependent. The requirements for expressing synchronisation constraints proposed by Bloom in [Bloom 79] seems to be broad enough for most purposes, though.

We summarise the information that may be relevant for synchronisation:

- ❏ Message properties, such as the message selector, message arguments. This may include the sender of the message and the message arrival time as well.
- ❏ The state of the instance variables[6].
- ❏ The synchronisation state of the resource, which means the state information that is specific to synchronisation. It includes information about the currently pending requests and the current activities within the object. One particular example would be the state of the message queue of the object.
- ❏ History information; this is information about previously fulfilled requests. For example, this could be the amount of requests of a certain type (cf. synchronisation counters [Robert 77]), or the type of the last served request.

It is not our intention to suggest that failure to meet all these aspects is a serious deficiency for any concurrent language or system. For instance, in a (soft) real-time system, information regarding the arrival time of a message and deadlines can be essential, whereas this is of no value in typical business applications.

It should be noted that the approaches based on behavioural abstraction that perform a *become* statement at, or after the end of methods (e.g. ACT++ [Kafura 89] and ABCL+ [Shibayama 91]) are obliged to predict the state of the object at the moment of message arrival. Thus they cannot deal with any information about arrived messages in the time between the *become* and the (test for) acceptance of a particular message.

3.2.7 Conclusion

This section is concluded with the necessary requirements for avoiding the occurrence of inheritance anomaly and a comparison with the analysis of inheritance anomalies made by Matsuoka et. al.

Requirements for Avoiding Inheritance Anomalies

From the sources of inheritance anomalies that were presented in this chapter, we directly derive a number of requirements to avoid the anomalies. These are all based on the structure of figure 3.2.10.

Synchronisation modularity: the specification of synchronisation must be completely separated from the method implementations. Further, it must be decomposable in a state abstraction function and a condition mapping function. The first covers the implementation specific aspects for abstracting the current state of the object into synchronisation conditions, and the latter provides a mapping from the synchronisation conditions (both local and inherited) to the acceptance of messages (both local and inherited). This corresponds to the structuring described by figure 3.2.10.

---

6 Although not supported by all languages, in the general case instance variables can be first-class objects themselves.

Synchronisation granularity: fine-grained synchronisation specifications are necessary to facilitate the reuse of -parts of- the specifications. Firstly a synchronisation constraint must be polymorphically applicable to additional messages. Secondly multiple synchronisation constraints must be composable to form one new synchronisation constraint in a descendant class. This is necessary both for single inheritance and multiple inheritance of synchronisation. Synchronisation granularity is mainly concerned with an incremental specification of the mapping *CM* in figure 3.2.10.

Expressiveness for Synchronisation Conditions: the expressive power that is available for specifying synchronisation conditions (i.e. the mapping *SA* in figure 3.2.10) must be sufficient so that all important synchronisation criteria can be expressed by it.

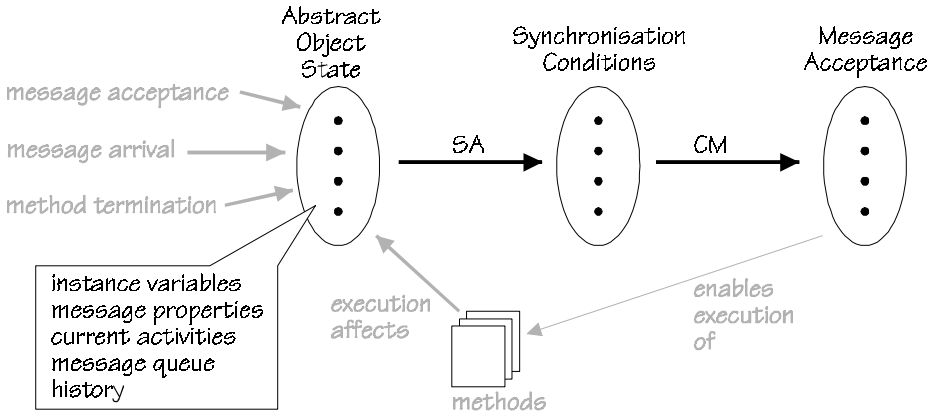


Figure 3.2.10 The generic structure of synchronisation schemes.

These requirements can be used as design criteria when developing new synchronisation schemes. Existing synchronisation schemes can be evaluated by judging to what extent they fulfil the criteria. Note that the general framework can model a wide range of synchronisation schemes. The design space for synchronisation schemes that fulfil all the requirements is restricted, but still a variety of mechanisms can be imagined.

Comparison with Analysis of Matsuoka et al.

The pioneering work in [Matsuoka 90] and [Matsuoka 93a] presents a different analysis of the inheritance anomaly. Their analysis is focused on, but not specific to, a model with behavioural abstraction. This assumes (a) that the state of an object is described by a set of mutual excluding states, i.e. an object can be in one state only, and (b) that with each state the set of all messages that are acceptable is associated.

Three classifications of the anomaly are distinguished:

- *State Partitioning*: The introduction of a new method in a subclass is likely to require a more specialised synchronisation, resulting in the specialisation of one of the states: that state will then be partitioned in two sub-states. The problems this will cause are due to the lack of synchronisation modularity; when the methods (or the body) of the superclass explicitly deal with the synchronisation, these must be modified in order to deal with the newly introduced states.
- *History-only Sensitiveness of Acceptable States*: This is a rather specific example, demonstrating that in some cases the synchronisation must be expressed in terms of the

history of the object. It is an example of the expressive power for synchronisation conditions.

- *Modification of Acceptable States*: In a subclass the acceptable states of inherited methods can be affected by new (or inherited) behaviour of the object; a state that allowed a message *m* may be modified into two substates, where *m* is not allowed any more in one of these substates. If it is not possible to compose the existing synchronisation specification with additional constraints, that can be either inherited or newly specified, this state modification requires redefinition of the synchronisation conditions of the superclass.

This categorisation gives an intuitive presentation of *what* goes wrong in certain categories of anomalies, whereas we focus more on *why* things go wrong.

3.3 The Composition Filters Approach

In this section the creation of concurrency and the specification of synchronisation constraints in the composition-filters model is explained and illustrated with a number of examples.

3.3.1 The Approach

Before proposing a mechanism for integrating concurrency and synchronisation in the composition-filters object model, the important design criteria are reconsidered, and it is explained how these are reflected in our solution. The three most important criteria are (1) integrate with composition-filters model, (2) avoid inheritance anomalies and (3) adopt the criteria proposed in section 3.1.

Integration with the Composition-filters Model

The composition-filters model attempts to separate domain-specific aspects as much as possible into distinct modules, by concentrating them in filter types. This has two important advantages: (a) particular features, in this case concurrency and synchronisation, can be completely ignored when not required, and (b) the various domain-specific features are orthogonal to each other, allowing them to be combined without interference[1].

Thus, the introduction of new filter types provides a clean method for extending the computation model with new features, without violating the current semantics of the object model. As far as extensions are not concentrated in new filter types, they should be as much as possible transparent to the programmer.

An important consequence of the integration within the composition-filters model are the semantics of inheritance: the delegation-based inheritance mechanism supported by the composition-filters model does not merge the code of the classes in the inheritance hierarchy, but creates fully-fledged instances for each of the ancestor classes and does not weaken their encapsulation. It is obvious that in this approach the synchronisation for a method that is defined by one of the ancestor classes cannot be weakened in a subclass, as messages are subject to the synchronisation defined by the ancestor instance, before they can be executed.

Although this limits the freedom in defining synchronisation constraints for inherited methods, it corresponds to good software-engineering practice: when relaxing synchronisation constraints of an inherited method consistency can never be guaranteed. Therefore the freedom in concurrent access offered by the class that defines the method can only be restricted in subclasses. A similar argument is presented in [Frølund 92].

---

1 This does not mean that each filter of an object is completely independent of the other filters, but that filters that are not contradicting can be freely merged. In general, a message is subject to all the constraints and manipulations imposed by each filter of an object.

Avoid Inheritance Anomalies

Because one of our primary goals is to support the construction of large systems through reusable and extensible software components, avoiding the occurrence of inheritance anomalies is an important aspect. Based on the criteria in 3.2.7 the following design decisions for the synchronisation scheme are taken:

- Synchronisation modularity: the synchronisation scheme must consist of two separate components: a state abstraction function, which maps the abstract state of the object to a synchronisation condition, and a condition mapping that associates the synchronisation conditions with messages. The state abstraction function is implemented with the *conditions* of the composition-filters model, the condition mapping is defined by a filter specification.
- Synchronisation granularity: By separating and labelling -synchronisation- conditions, they can be referred to and applied to different messages (in subclasses as well), thereby obtaining polymorphically applicable synchronisation constraints. Composition of synchronisation constraints can be done with AND semantics; i.e. the constraints of all components must be satisfied. This is achieved by specifying multiple filters. If this is too restrictive, the condition mapping can be redefined (without violating constraints in superclasses, though). Open-ended synchronisation specifications are supported as well through the wild card mechanism in filters.
- Expressiveness for synchronisation conditions: the conditions in the composition-filters model have the full expressive power of message expressions, only restricted by the constraint that they must be side-effect free. The properties of the message are available through a pseudo-variable, and information regarding the synchronisation state of the object is available as well.

Criteria for COOPLs

The important properties of the object model that we adopt are that it supports only active objects, allows for intra-object concurrency and nesting of active objects. Strong encapsulation is maintained, neither client objects nor subclasses can access the internal data structure of an object. Through the encapsulation of synchronisation, the mechanisms for reuse and composition of -sequential- objects apply to concurrent objects as well. Separate reuse and composition of synchronisation specifications requires the labelling of synchronisation specifications and making these labels visible on the interface of objects[2]. The aspect of efficiency is only dealt with to the degree that the synchronisation mechanism is defined such that it has a straightforward computation.

3.3.2 Creating Concurrency

The creation of concurrency in a composition-filter program is achieved by adopting the message passing semantics of the non-blocking RPC. This maintains the request-reply model for communication; a client cannot distinguish a blocking RPC invocation from a non-blocking invocation. However, the server object can issue an *early return*, which replies the

2 This is already possible for the behaviour of objects, which is visible through the names of the messages that the object supports.

result of the invocation to the client object, while continuing executing the remaining statements of the method.

We illustrate this with a method that is defined for a buffer object. This method is named killBefore and takes a single argument. The method searches the buffer, starting with the newest element, to find an element that equals the argument. Success of this search returned as a reply to the client. This causes the blocked thread of the client to continue. At the same time the killBefore method continues to remove all the elements in the buffer that are older then the located element (provided the latter was available, of course). At the end of the method, this thread is terminated.

```
ClientObject
    ...
    clientMethod(..)
    begin
        ...
        // all elements that were put
        before
          the first element '10' are
        removed
        if aBufferObject.killBefore(10);
        then self.print('cleaned up')
        else self.print('no 10 found');
        ...
    end;
```

```
ABufferObject
    ...
    killBefore(value:Integer) returns
    Boolean
    begin
        // search for <value>
        return /* <value> found */;
        // remove rest of elements, if
            available
    end;
```

In general, a method can be divided in two parts: the part before the return and the part after the return statement. When there are no more statements after the return, the method corresponds to a conventional message invocation. When there are no statements before the return, the semantics of calling the method are similar to synchronous one-way message passing; the client can immediately resume its execution, after the message has been accepted by the receiver object. The obvious difference is that it is possible to return a reply value as well.

A problem appears when a method executes more than one return statement: this is possible because after the first return statement the thread can continue. In this case an error will occur, since it is not possible to return multiple replies for a single message invocation. To avoid such problems and structure the method implementation, it is good practice to use only a single return statement in a method body.

One particular application of the early return statement is within the initial method: when a thread initiates the creation of new object, after the creation of the nested objects, the initial method is executed. Meanwhile, the creating thread is blocked, which ensures that the new object cannot be referred to by any other object, and thus no messages can be sent to it. By placing an early return statement in the initial method, the thread that created the object can continue while the initial method continues as well. This can for instance be applied for defining a process that executes 'in the background' during the lifetime of the object[3].

[3] We intend to investigate the possibility of killing such (active) processes when the object is garbage collected. The usual approach is not to garbage collect an object when it is still referred to by a process (see also [Kafura 90b, 91]).

This is exemplified with the following example of a clock object:

```
class Clock interface
    comment implements a 24-hour clock object;
    externals
        systemTimer : Timer;   // assume an interface to a hardware clock
    methods  // for example
        curMinute returns Integer  comment returns current minute value;
        curHour returns Integer  comment returns current hour value;
        ...
    inputfilters
        disp : Dispatch = { inner.* };
end // class Clock interface

class Clock implementation
    instvars
        hour, minute : Integer;
    initial
        begin
            hour := 12; minute := 0; return;   // return without an argument returns nil
            while true do  begin systemTimer.waitMinute; server.incrMinute; end;
        end;
    methods
        incrMinute returns Nil comment "increments minutes, and hours when necessary"
end // class Clock implementation
```

The initial method first initialises the instance variables and then issues a return statement. After the return statement an infinite loop is started, which waits until another minute has passed and then updates the state of the object.

It must be noted that the application of an early return, in particular in the initial method, may not combine with mutual exclusive objects. This aspect will be touched upon later in this section.

3.3.3 Synchronisation with Wait Filters

We now continue to explain how the actual synchronisation of messages is integrated in the composition-filters model. For the sake of brevity, we focus on the messages that are received by an object, but the discussion is fully applicable to both input and output filters.

The Object Manager

The responsibility for handling messages and evaluating filters lies with the *object manager*. This is a system-defined object that handles the reception of messages, their evaluation by filters, message-dispatch, and the scheduling of messages as well. In addition, an object manager maintains information about the received messages, active threads, etcetera. The object manager is a first-class encapsulation of a part of the virtual machine, and can be accessed from within the object through message invocations.

The reception, filtering and synchronisation of messages thus all takes place within the object manager, as illustrated in figure 3.3.1. This figure also demonstrates how synchronisation of messages is achieved: we introduce a new type of filter, *Wait*, that can block messages. The blocked messages are put in a message queue, and can pass the filter only when they are accepted for execution and removed from the queue. A message arriving

at a *Wait* filter is evaluated according to the filter specification, resulting in either acceptance or rejection of the message. In the first case, the message can immediately proceed to the next filter. Optional message substitutions are ignored by wait filters. When the message is rejected by the filter, it will be put in the queue, and removed only once the evaluation of the message against the filter specification does result in acceptance[4].

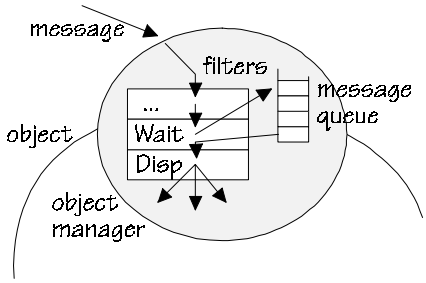


Figure 3.3.1 The manipulation of messages by the object manager

An important property of this mechanism is that the composition-filters model and filter specifications can be applied consistently, including for the specification of synchronisation constraints. The details of synchronisation specifications are discussed further in this section.

The concurrency model of the composition-filters model supports internal object concurrency because the number of active threads within an object is not restricted by the model itself, nor does the concurrency model impose any synchronisation constraints. The consistency of an object must be guaranteed through definition of the appropriate synchronisation constraints in *Wait* filters.

By default, the pre-processor includes for every object a filter, defSync, that provides mutual exclusion, not accepting requests while there is still an active thread within the object. Mutual exclusion ensures that no inconsistencies will occur in the values of the instance variables due to concurrent access. By specifying a compiler option, the default filter for mutual exclusion can be excluded, allowing less restrictive concurrency constraints to be specified.

The scheduling of requests is the responsibility of the object manager. The object manager ensures that the following properties hold:

- ❏ Messages are received indivisibly: only one message is dealt with at a time, which imposes an order on the received messages. This order is reflected in the message queue.
- ❏ Filter evaluation is atomic: the object manager evaluates messages one by one. This ensures that two competing messages will not inadvertently be accepted both.

---

[4] In this presentation of the scheduling of received messages it is assumed that no other messages -that were received earlier and remain in the message queue- are acceptable for activation at that instance. Otherwise a newly received message might inadvertently be accepted earlier than 'older' messages in the queue.

- When two messages are subject to the same constraints, the message that is closer to the head of the queue will be dispatched first[5]. This does not mean that messages are always executed in first-come-first-served order. A message that arrived earlier but does not yet fulfil the conditions imposed by the filters will remain blocked and allow the execution of another message, provided that message does fulfil its conditions.

The object manager is also responsible for the re-evaluation of wait filters to enable blocked messages to resume execution. Re-evaluation of a wait filter occurs at least at every change of the object manager state. Thus, whenever the state of the object changes such that a blocked message can be unblocked, the message will be accepted before or immediately when the state of the object manager changes. Examples of changes in the object manager state are message arrival, dispatching, method termination, etcetera.

As discussed previously, a synchronisation scheme should consist of two separate parts: a state abstraction function and a condition mapping function. It will now be described how these are specified in the composition-filters model.

State Abstraction with Conditions

The condition mapping function is achieved through the specification of the conditions of the composition-filters object model. These implement a mapping from the abstract state of the object to a finite set of conditions; with each of these a boolean value is associated that designates its validity. We use the bounded buffer to illustrate the application of conditions:

```
class BoundedBuffer(limit:Integer) interface
    comment implements a bounded buffer with synchronisation;
    conditions
        Empty; Partial; Full;
    methods
        put(Any) returns Nil;
        get returns Any;
    inputfilters
            // a synchronisation filter will be included here
        disp : Dispatch = {inner.* };
end // class BoundedBuffer interface

class BoundedBuffer implementation
    instvars
        store : Array(limit);  // index ranges from 1 to <limit>
        head, tail : Integer;
    conditions
        Empty begin return head=tail end; // no elements in the buffer
        Partial begin return (inner.Empty.not and inner.Full.not) end;
        Full begin return (limit+tail-head).mod(limit)=1;  end;  // the buffer is full
    initial
        begin head:=1; tail:=1; end;
```

5 One approach to realising this is to make a snapshot of the state of the object, or of the values of the conditions, before iterating over the messages in the queue. The iteration must then start from the oldest and end with the latest message.

```
    methods
        put(elem:Any) returns Nil
            begin store.atPut(head, elem); head:=head.mod(limit)+1; end;
        get returns Any
            begin return store.at(tail); tail:=tail.mod(limit)+1; end;
        // other methods, e.g. for allowing direct access to the buffer by subclasses
end // class BoundedBuffer implementation
```

This example demonstrates how conditions can be used to abstract the state of the object in a similar way as is done in behavioural abstraction approaches: the state space is subdivided into three states, Empty, Partial and Full. However, the programmer is responsible for ensuring that these abstract states do not overlap (and cover the entire state space of the object). This is not a necessary requirement, though. An alternative condition definition would be the following:

```
...
conditions
    ContainsElements begin return head<>tail end;
    SpaceLeft begin return (limit+tail-head).mod(limit)>1 end;
    ...
```

This shows that the state spaces defined by a condition are not necessarily mutual exclusive. We would like to make two further observations. The first is that allowing the full expressive power of arbitrary message expressions for defining conditions is not overdone, as even in these relatively simple examples various operations are used. The second observation is that it is possible to refer to another condition in the definition of a condition, as exemplified by the Partial condition.

Condition Mapping with Filters

The second part of the synchronisation scheme is the mapping from the synchronisation conditions to the accepted messages for that condition. We introduce a new filter type for expressing this mapping. The type is labelled Wait and has the following intuitive definition: when a message arrives at a *wait* filter, it can only proceed when the message is accepted by the filter, and it will be blocked otherwise, until the message *can* be accepted by the filter.

Consider the following input filter specifications for the bounded buffer example:

```
inputfilters
    bufferSync : Wait = { Empty=>put, Partial=>{put, get}, Full=>get };
    disp : Dispatch = { inner.* };
```

The first filter is of type Wait; it defines when a message is accepted. This exactly corresponds to a mapping from the conditions Empty, Partial and Full to the messages put and get. The "=>" symbol accepts the messages on its right hand side when -one of- the conditions on the left hand side is satisfied. This also allows for different styles for expressing the synchronisation of the bounded buffer:

```
inputfilters
    bufferSync : Wait = { {Empty, Partial}=>put, {Partial, Full}=>get };
    disp : Dispatch = { inner.* };
```

The same synchronisation constraints can be expressed as follows (with the alternative set of conditions that were previously defined):

```
inputfilters
    bufferSync : Wait = { ContainsElements=>get, SpaceLeft=>put };
    disp : Dispatch = { inner.* };
```

This mapping resembles a guard-based specification, as for each message that is currently defined by the class, there is a single condition. When new methods are added to the class or one of its subclasses, additional messages can be associated with the conditions, though.

To illustrate the synchronisation of messages, we show the message queue for an instance of class BoundedBuffer that subsequently receives the messages put, get, get', get'' and put'. We assume the following input filters and condition implementations for class BoundedBuffer:

```
inputfilters
    bufferSync : Wait = { Empty=>put, Partial=>{put, get}, Full=>get };
conditions
    Empty begin return head=tail  end; // no elements in the buffer
    Partial begin return (innerEmpty.not and inner.Full.not)  end;  // not empty, not full
    Full begin return (limit+tail-head).mod(limit)=1;  end;  // the buffer is full
```

The arrival, buffering and dispatching of these messages is shown in figure 3.3.2.

As the first message put is received (no. 1), it is matched against the bufferSync wait filter. The put matches with the first filter element, as the Empty condition is valid. Thus the message is immediately dispatched and therefore removed from the queue again (no. 2). Now the buffer contains one element, causing condition Partial to become true. So when message get arrives (no. 3), it is accepted by the filter, and immediately dispatched (no. 4). Condition Partial is no longer valid, causing the received message get' to be blocked (no. 5). When message get'' subsequently arrives, it is placed in the queue, after get', as well (no. 6).

When put' arrives, it will be placed at the tail of the queue (no. 7), but since it is the first acceptable message in the queue, it will be dispatched prior to the get messages (no. 8). After put' has completed its execution (as was mentioned before mutual exclusion is enforced by default), condition Partial is valid once again. This enables both get' and get'', in which case the first applicable message (get') in the queue is dispatched. After the execution of get', the condition Empty will become true and condition Partial will be false, therefore get'' remains in the queue (no. 9).

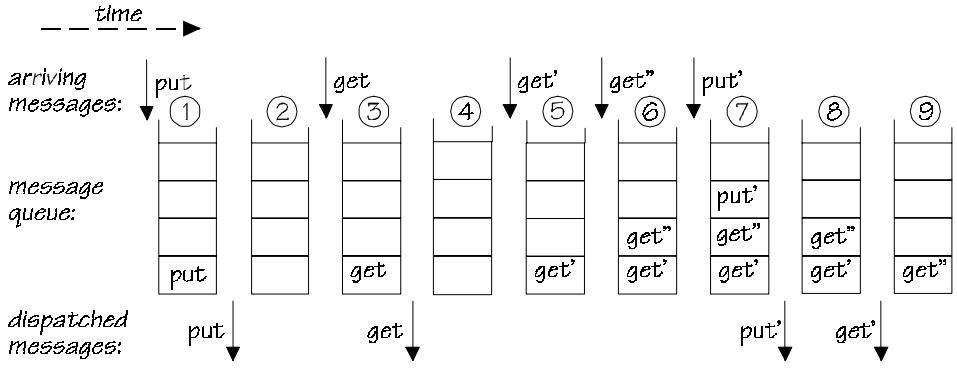


Figure 3.3.2 Example of message scheduling in a bounded buffer.

The composition-filters model allows the specification of Wait filters in both input and output filters. The synchronisation of outgoing messages does not appear frequently in the examples. The most important reason is that the object-oriented paradigm emphasises a client-server model,

where the server object receives a message and determines what to do with it: reject it, accept it, or block for a while until the object is ready to serve the request. In this approach synchronisation by the client (i.e. the object that sends the message) is not appropriate. There are some situations, though, where synchronisation in output filters can very useful. One typical example is when the client object can, or should not perform the synchronisation. This can be for modelling reasons, or when using predefined objects from a library that do not provide the desired synchronisation. This should not be a problem as long as the client is not taking over the responsibilities of the server object.

Multiple Wait Filters

The definition of multiple wait filters provides a means for composing synchronisation specifications. This serves two purposes: firstly, it allows for decomposing complex synchronisation specifications into smaller parts that are easier to develop, maintain and reuse, and secondly, the composition of synchronisation specifications is an important property for effective reuse, and therefore necessary to avoid certain inheritance anomalies.

As an example, we add a method get2 to the BoundedBuffer class. This is a get operation that removes two elements from the buffer. One of the approaches to defining synchronisation would be to restructure the Empty/Partial/Full conditions, but, in particular because we want a modular synchronisation specification, it is simpler to define a guard for the new method with the following filter:

```
get2Sync : Wait = { TwoOrMore=>get2, True=>{get, put} };
```

The implementation of condition TwoOrMore is as follows:

```
TwoOrMore begin return (tail-head+limit).mod(limit)>2 end ;
```

The filter specification reveals an important issue, though: because all the messages to the object -normally- pass all the filters, the wait filter specification must also specify synchronisation for all messages, even when this is no synchronisation at all. In the get2sync filter this appears as the second filter element, "True=>{get, put}". The exclusion operator can be applied to avoid having to specify all the other methods of the object (including the inherited ones) one by one:

```
get2Sync : Wait = { TwoOrMore=>get2, True~>get2 };
```

This means that all messages except the get2 are always accepted by the get2sync filter.

However, a similar argument applies to the bufferSync filter that defines the synchronisation for the get and put message: in the current definition this will never match a get2 message, which will thus block forever. The same applies for the methods that are inherited by default from Object. Therefore, it is good design practice to associate a True condition with all the messages that the synchronisation is not concerned with[6]. The bufferSync filter is accordingly changed to:

```
bufferSync : Wait = { Empty=>put, Partial=>{put, get}, Full=>get, True~>{get, put} };
```

---

6 An additional advantage is that messages that the object does not support will pass the filter as well, and probably cause an error later, e.g. in the Dispatch filter, instead of remaining blocked forever. This is also solved with an error filter as the first filter in the set that rejects all unknown messages.

Note that the definition of multiple filters specifies an AND constraint between the separate synchronisation specifications, as each message must pass both filters. In principal, with each wait filter a separate message queue is associated. However, this can lead to awkward and unintuitive problems due to the time delay between the filter evaluations. This is illustrated with the following example filter definition that is applied to the bounded buffer class:

```
inputfilters
    memorySync : Wait = { MemAvail=>put, True~>put };
    bufferSync : Wait = { Empty=>put, Partial=>{put, get}, Full=>get, True~>{get, put} };
    dispatching : Dispatch = { inner.* };
```

For this example we assume that in a particular application each object has a limited amount of memory available for storing elements in the buffer, this may change during the life-time of the application. Therefore in this specification an additional constraint is defined for put messages: they are only allowed while there is still enough memory available, otherwise they must be blocked. Now consider the following sequence of events:

(1) A put messages is received while the object is out of memory.

(2) The processing of the message at the memorySync filter causes it to be blocked.

(3) After a while, the object regains some memory, and the put message is enabled.

(4) The put message passes on to the bufferSync filter and is blocked there because the buffer is full.

(5) Meanwhile, the object runs out of memory again...

(6) Due to the execution of one or more get messages, the out message is enabled again

(7) The last filter, dispatching, causes the execution of the put message.

The problem here is that the execution of the put message is initiated while the object has insufficient memory available. The constraints defined by the first wait filter are 'forgotten' once the filter is passed.

The solution to this problem is offered by the following extension to the semantics of the wait filters: subsequent wait filters in the filter set of an object are merged, they share a single message queue, and a message is only removed from the queue when *all* constraints by *all* the subsequent wait filters that apply to the message are satisfied.

For the previous example this means that the synchronisation constraint of the put message is defined as: (MemAvail $\wedge$ (Empty $\vee$ Partial)), whereas the synchronisation of the get is defined as (True $\wedge$ (Partial $\vee$ Full)). This avoids the problem that we just described.

It may seem a bit awkward to have multiple message queues in the set of input filters, but there are at least two important applications of this: firstly, consider a filter set with the following structure:

```
inputfilters
    sync1 : Wait;       // synchronise messages before redirected to ACT
    sync2 : Wait;
    toACT : Meta;       // the ACT manipulates or rearranges messages and fires them again
    sync3 : Wait;       // synchronise messages after they return from ACT
    disp : Dispatch; // dispatch the messages
```

In this filter set, between the sync2 wait filter and the sync3 wait filter the message can be modified and delayed by the ACT. It would be inappropriate to require that the constraints of the sync1 and sync2 filters are to be satisfied -again-.

The second reason for supporting the multiple message queues is that this can be used as a technique to implement certain synchronisation constraints, in particular with respect to message arrival time. This is can be used as an alternative to rearranging the order of messages in the queue, because we do not allow queue manipulation. An example of this is in the implementation of the readers/writers problem with equal priority (in section 3.4).

Synchronisation Counters

To manage the amount of concurrency in an object, it is necessary to collect information about the status of the object: how many threads are active within the object, how many pending requests, etcetera. This information is maintained by the object manager of each object. An object can refer to its object manager through the pseudo-variable "^self", and to the object manager of the server object through "^server". The object manager provides a number of methods for retrieving information.

The information is based on *synchronisation counters* (see e.g. [Robert 77], [Gerber 77]): these are maintained by the object manager and count the number of times certain events occur. Examples of these events are the arrival, acceptance, execution and termination of messages. Synchronisation counters are also adopted in Guide [Decouchant 91] and *behavior counters* [Neusius 91b].

We distinguish two categories of events: the first category deals with the events caused by messages that are processed by (wait) filters, the second category deals with the initiation and termination of local method executions. The first table defines the methods provided by the object manager that deal with the message processing in filters:

Filtering counters:	
in(filterId)	The number of messages received at filter <filterId>
inFor(filterId, messageId)	The number of messages <messageId> received at filter <filterId>
out(filterId)	The number of messages that passed filter <filterId>
outFor(filterId, messageId)	The number of messages <messageId> that passed filter <filterId>

In the second table the synchronisation counters that describe the local execution (and termination[7]) of methods are made available through the interface of the object manager:

7 Note that the counters presented here cannot detect the termination of delegated messages. This is because once a message is dispatched to an internal or external object, the dispatching object is not concerned with the message anymore.

Execution counters:	
started	The number of started local methods.
startedMethod(methodId)	The number of started local methods <methodId>.
done	The number of terminated or aborted method executions.
doneMethod(methodId)	The number of finished <methodId> methods.

For convenience, the most important expressions based on synchronisation counters are provided as separate methods of the object manager. We call these 'derived counters':

Derived counters:	
blocked(filterId)	The number of messages currently blocked at <filterId> (= in(filterId)-out(filterId)).
blockedFor(filterId, messageId)	The number of messages <messageId> that are currently blocked at filter <filterId>.
blockedReq	The sum of all currently blocked requests in all message queues of the input filters.
blockedReqFor(messageId)	The sum of all currently blocked requests <messageId>.
blockedInv	The sum of all currently blocked message invocations in all message queues of the output filters
blockedInvFor(messageId)	The sum of all blocked invocations <messageId>
active	The number of currently executing local methods (= started-done)
activeMethod(methodId)	The number of currently executingmethods <methodId>

Although the bookkeeping of all these synchronisation counters might incur significant overhead, this can be avoided by maintaining only those counters that is referred to within the implementation of objects.

Note that not every message that arrives at an object results in the execution of a local method: the message may bounce, or be delegated to an internal or external object. Examples demonstrating how these synchronisation counters can be applied will appear later in this chapter.

3.3.4 Extension of a Concurrent Class

We will now demonstrate the reuse and extension of classes that incorporate synchronisation specifications. We use example of introducing a get2 method again, this time added to a subclass of BoundedBuffer, called Buffer2. The get2 method retrieves two elements instead of one element from the buffer, and therefore requires an adapted synchronisation specification.

```
class Buffer2 interface
    internals
        buf : BoundedBuffer;
```

```
    conditions
        buf.*;  // to maintain the synchronisation interface: discussed later in this subsection
        TwoOrMore; // new condition, defined in the implementation part
    methods
        get2 returns Tuple;   // we assume a predefined class <Tuple> that stores 2 objects
    inputfilters
        sync : Wait = { TwoOrMore=>get2; True~>get2 };
        disp :Dispatch = { inner.*, buf.* };
end // class Buffer2 interface

class Buffer2 implementation
    comment "the implementation of the condition <TwoOrMore> and the method <get2>"
    conditions
        TwoOrMore  begin return buf.size>=2; end;  // the size of the buffer must be
    accessible
    methods
        get2 returns Tuple
            temps pair : tuple;
            begin pair.init(server.get, server.get); return pair end;
end // class Buffer2 implementation
```

Defining inheritance from BoundedBuffer is done in the same manner as introduced in the previous chapter. The internal buf represents an instance of class BoundedBuffer, and the dispatch filter specifies that all methods defined by *inner* are executed locally and all methods defined by BoundedBuffer are delegated to the buf internal object.

The synchronisation specification of Buffer2 is incremental; only for the additional methods need the synchronisation constraints be specified, all other messages are not constrained. This is expressed with the sync wait filter: "{ TwoOrMore=>get2; True~>get2 }". Thus the get2 message is accepted only if the TwoOrMore condition is valid, which means that there are two or more elements in the buffer.

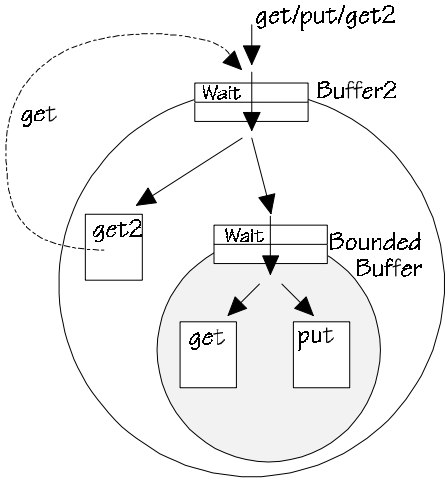


Figure 3.3.3 The structure of the Buffer2 example.

Figure 3.3.3 illustrates the extension of a concurrent class: the inherited methods that are defined by BoundedBuffer are synchronised at the internal 'superclass' object. The subclass need not bother with the synchronisation of the inherited methods (as this is 'inherited' as

well). However, the subclass *may* define additional synchronisation constraints, as all the inherited messages must pass its filters as well.

Inheritance of synchronisation thus has the following important properties:

1. Synchronisation can be specified incrementally; the synchronisation constraints of the new methods in the subclass can be specified independently from the synchronisation of the methods of the superclass.
2. The synchronisation constraints defined for the methods of the superclass can never be weakened. It is only possible to define additional constraints on the inherited methods.

The importance of maintaining the (synchronisation) interface of an object is illustrated through an example that again extends the Buffer2 class. This is a class Buffer2Latest, that adds a method getLatest, which is similar to the get, but retrieves the latest added element, i.e. LIFO behaviour instead of the FIFO behaviour of the get. The interface definition of the class is as follows:

```
class Buffer2Latest interface
    internals
        buf2 : Buffer2;
    conditions
        buf2.*;          // we make the conditions on the interface of buf2 available!
    methods
        getLatest returns Any;
    inputfilters
        sync : Wait = { {Partial, Full}=>getLatest; True~>getLatest };
        disp :Dispatch = { inner.*, buf2.* };
end // class Buffer2Latest interface
    // the implementation part, consisting of the definition of the getLatest method is omitted
```

The synchronisation of the getLatest method is exactly the same as the synchronisation of the get. The conditions that are required to specify this synchronisation are defined by the BoundedBuffer class. It is thus important to make the conditions available to the subclasses. Therefore, even when a class does not use the conditions defined by its parents, as is the case for Buffer2, it is important to make these conditions available on the interface so that they can be used in subclasses. This is achieved by declaring "buf2.*" in the conditions clause of the interface, which makes all the conditions on the interface of buf2 available.

This is essentially the same as the transitive inheritance of methods along the inheritance hierarchy. By making this explicit, it is possible to redefine conditions and to solve naming conflicts when multiple parents define conditions with the same name.

3.3.5 Default Synchronisation

In chapter 2 we described the default behaviour of objects. The default behaviour is provided for the convenience of the developer, and can be turned on or off using a compiler option. As the default synchronisation, mutual exclusion is adopted. The description of the default synchronisation also serves as an example how a typical synchronisation problem can be expressed with wait filters.

Mutual Exclusion

Mutual exclusion is a generic technique to protect an object against data inconsistency due to concurrent access. It means that we allow only a single method execution within an object at a time. The important advantage of enforcing mutual exclusion at the interface of the object is that it allows for ignoring concurrency in the implementation of the object while the object can still be used safely in a concurrent environment.

In the composition-filters model a distinction is made between the local execution of a method and the execution of inherited and delegated methods, because the latter are encapsulated within the internals respectively externals of the object. This has the advantage that they may be processed concurrently without causing data inconsistency as they all execute within a separate encapsulated object.

Therefore, mutual exclusion needs only be enforced on the execution of local methods. This is achieved by the following synchronisation specification (consisting of a wait input filter and a condition implementation):

```
inputfilters
    ...
    defSync : Wait = { Free=>inner.*, True~>inner.* };
    ...
conditions
    ...
    Free  begin return ^self.active=0 end ;
    ...
```

The defSync wait filter defines a synchronisation constraint for all the messages that correspond to local methods of the object.[8] Only when condition Free is satisfied, these messages are allowed to continue. This is the case when the number of local methods that are executing equals zero. All other messages can immediately pass the filter. Thus, as long as there is still an active thread inside the object, no other local methods can be executed.

Allowing the execution of inherited and delegated messages in parallel with local methods brings a number of advantages but has some drawbacks as well. The advantages are:

- It can increase the amount of parallelism in an application.
- A message invocation that is delegated and then blocked at the delegated object does not prohibit other (inherited) messages to be accepted. This may be necessary in order to bring the delegated object into another state such that the blocked invocation can be accepted after all. This is also the case when a local method invokes (directly or indirectly) a message of a delegated object.

This also allows for avoiding a problem explained in [Bal 92] that occurs when a message to a nested object is blocked. If the message is sent by a mutual exclusive object, deadlock will occur. This could also occur when dispatching to an internal object would block the encapsulating object, as illustrated in the figure below:

8 The filter element "inner.*" selects all the messages that are locally defined. In the -unlikely- case where a local method is not made available on the interface of an object, the defSync filter may need to be redefined to adopt to the desired synchronisation.

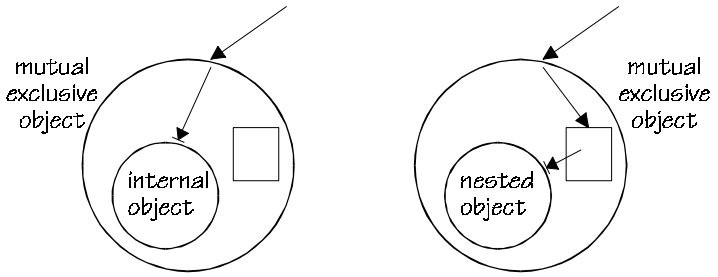


Figure 3.3.4 Blocking invocations on encapsulated objects.

By always allowing access to the encapsulated object through the dispatch mechanism of the encapsulating object, the state of the encapsulated object can be modified so that the blocked thread can be enabled again. Although the problem described by Bal can still occur when the nested object is an instance variable, this can be tackled by relaxing the mutual exclusion constraint of the encapsulating object, or when the nested object encapsulates internal threads.

There are a number of potential problems, though, that can occur due to the relaxation of the mutual exclusion:

- When an inherited method invokes a message on *server*, there may have been other requests accepted and executed by the *server* object, interleaving the execution of the inherited method. This may in some circumstances be undesirable, and can then be avoided by adding an additional synchronisation constraint that blocks all message acceptance by the *server* object during the execution of the inherited method.
- Consider a method transaction that subsequently invokes several methods defined in its superclass to read and modify the state of the superclass object. To avoid that these invocations are interleaved with other messages directly inherited from the superclass, additional synchronisation constraints must be defined.
- Assume a subclass of the BoundedBuffer example, Buffer22. This class defines the methods put2 and get2, that are both implemented by calling the put respectively get methods of the superclass two times. When a get2 message followed by a put2 is sent to an empty Buffer22 instance, the nested get invocations in the body of get2 will block the object, prohibiting the execution of the local method put2. Through proper specification of synchronisation for the inherited put and get messages this problem can be solved, as will be demonstrated later in this chapter.

The rigorous synchronisation constraints imposed by mutual exclusion may lead to deadlock in certain situations. The default synchronisation we proposed relaxes the constraints, but this may introduce new problems that are due to the distribution of the synchronisation constraints over inherited and inheriting object. These problems can be resolved by reproducing the synchronisation of the superclass methods in the subclass.

Recursive Messages

One of the problems with mutual exclusion synchronisation is that recursive calls to *self* and *server* could normally block, as there is already an active thread within the object. This problem was also identified in [Briot 87] and [Yokote 86]. Supporting recursive calls is necessary for two reasons. First, the pseudo-variables *self* and *server* allow for calling methods defined in superclasses or subclasses respectively. As inheritance is achieved through a dispatch filter, a message sent to *self*, for example, must necessarily pass the

filters of the object. The second reason is that the methods that are called during the execution of an accepted message may need to synchronise again.

Thus, recursive messages must be subjected to evaluation by wait filters and may be blocked as well. But a recursive message should not be blocked because of the mutual exclusion synchronisation constraint. This problem is resolved by redefining the mutual exclusion filter so that it can deal with recursive messages:

```
defSync : Wait = { {Free, Recursive}=>inner.*, True~>inner.* };
```

The inner methods are now allowed to execute when either the Free or the Recursive condition is satisfied. The Recursive condition is true only when the currently evaluated message is a recursive message. The recursiveness property can be obtained from the message representation, as shown in the condition implementation:

```
Recursive  begin  return message.isRecursive;  end;
```

All messages that are sent to the pseudo-variables *self*, *server* or *sender* are recursive. Note that messages sent to *inner* (i.e. direct calls on local methods) are not subject to filter evaluation, and thus will never block.

Mutual Exclusion vs. Early Returns

The definition of mutual exclusive synchronisation prohibits the concurrent execution of message requests on the interface of the object. Consider the following two example methods that are defined for the same object:

```
m returns Nil;
begin
    ...
    self.spawn(10);
    self.spawn(5);
    ...
end ;
```

```
spawn(n:Integer) returns Nil;
begin
    return;  // returns nil
    while n>0 do    // repeat n times
        /* perform some action */
end ;
```

Whenever method m is executed, it will call the method spawn. Because the target of the message invocation is *self*, this is a recursive message and it will not be blocked at the interface. The spawn method immediately performs an early return, and proceeds with the subsequent while-loop, in parallel with the -now resumed- method m. Therefore, after the two method invocations on spawn, three concurrent threads can be active within the object, regardless of the mutual exclusion input filter!

We consider this form of intra-object concurrency not as a serious threat to the consistency of the object though. This is because the intra-object concurrency achieved in this way is only possible in the presence of recursive calls. It is thus the responsibility of the developer of the class to avoid inconsistencies due to concurrency. Because intra-object concurrency cannot be created in this way by calls from external clients, it should not cause consistency problems.

Inserting the Default Synchronisation

The default synchronisation will be inserted (when appropriate) within the set of input filters, immediately after the last wait filter defined for the object. The filter specification is defined by the internal object default. This makes it possible to adopt a different default

synchronisation by redefining class Object or replacing the internal default with another class.

```
inputfilters
    ...
    userSync : Wait = .. ;  // this is the last wait filter
    default.defSync;
    ...
```

As a reminder: default filters are not inserted when a filter with the same label is already defined by the object. This makes it for instance possible to override only the default synchronisation by defining another filter defSync, while retaining all other default object behaviour[9].

On the messages that are sent by the object, no synchronisation constraints are imposed. Thus, no wait filters are inserted in the set of output filters by default.

The primitive objects in the system, such as integers, booleans and strings all provide a behaviour that is equivalent to the default behaviour. Thus, an application programmer is not necessarily concerned with synchronisation issues, but can enforce mutual exclusion for all objects by inserting the default filters through a compiler option.

9 Although the rules regarding the default behaviour of objects may seem a bit complex, it must be noted that these are completely optional, and in fact aim at increasing convenience in the definition of classes.

3.4 Examples of Wait Filters Applicability

In this section a number of examples are given to demonstrate the applicability, expressiveness and extensibility of synchronisation through wait filters. The examples are divided into a number of categories. First it is shown how several variations of message passing semantics can be defined, in particular asynchronous message passing and future message passing. Then we use reader/writer synchronisation to illustrate how intra-object concurrency can be managed. Finally we show two extensions to the bounded buffer example, addressing more complex synchronisation. The latter are typical examples leading to inheritance anomalies in some other approaches.

3.4.1 User-defined Message Passing Semantics

We show two examples of adapted message passing semantics. In subsection 3.1.3 various message passing semantics were explained. In this subsection asynchronous message passing and future message passing are shown to be expressible with the composition-filters model. In both examples the object that sends the message selects these alternative message passing semantics explicitly per message invocation. Several variations to this can be realised, for example transparent modification of the message passing semantics through ACTs or server-based modification of semantics.

Asynchronous Message Passing

Asynchronous message passing is a useful tool for increasing the amount of concurrency in an application, because the client and the server proceed in parallel. It assumes that no result is returned by a request, corresponding to a nil result type in our model.

The example defines two classes: a class Aclient that serves as an example class, initiating asynchronous message invocations. To do this, it inherits the message sendAsync from class Async. By supplying the sendAsync message with the desired target, message selector and argument, an asynchronous message is defined and executed:

```
class Aclient interface
    internals
        mps : Async;   // (asynchronous) Message Passing Semantics
    methods
        ...
    inputfilters
        inh : Dispatch = {inner.*, mps.* };  // inherit methods of class Async
end // class Aclient interface
class Aclient implementation
    ...
methods
m() returns ...
    begin
        ...
        server.sendAsync( serverObject, 'request', 1000 );
        ...
    end;
end // class Aclient implementation
```

After the invocation "server.sendAsync(serverObject, 'request', 1000);" the subsequent statement can be -almost- immediately performed. Class Async implements the asynchronous semantics through the sendAsync message:

```
class Async interface
    comment "This class offers explicit asynchronous message sending through the
             message <sendAsync>. This message takes the target, message selector and
             an  argument, and asynchronously sends a message with these properties"
    methods
        sendAsync(Any, String, Any ) returns Nil;
    // default filters
end // class Async interface

class Async implementation
    methods
        sendAsync(rec:Any; sel:String; arg:Any) returns Nil
            comment for brevity, we do not deal with multiple arguments;
            temps mess : Message;  // creates a first-class representation of a message
            begin
                return ; // caller resumes execution with nil reply
                mess.setReceiver(rec);
                mess.setSelector(sel);
                mess.setArgAt(1, arg);
                mess.fire;
            end;
end // class Async implementation
```

Asynchronous message sending is achieved by first performing an early return, and then composing the actual message and sending it. After the receiver, message selector and argument of an instance of class Message have been defined, the first-class message is activated through the fire message.

Although the manner for specifying asynchronous message sending is a bit cumbersome, asynchronous message passing semantics can be fully simulated with this approach, comparable to e.g. ABCL that offers various message passing constructs.

Future Message Passing

The implementation of future message passing semantics is organised similar to the previous example: an example class, aClient', inherits from class Future, that provides a method sendFuture. This method immediately returns a proxy object, an instance of class Proxy, and then sends the message to the defined target. Thus, the client can proceed its execution, with the proxy object as the result of the future message. As long as the client does not access the proxy object, it can continue its activities even while the result of the prior future message invocation is not available. However, the messages that are sent to the proxy object will be blocked until the result of the invocation is available:

```
class Aclient' interface
    internals
        fut : Future;
    methods
        ...
```

```
    inputfilters
        inh : Dispatch = {inner.*, fut.* };  // inherit method(s) of class Future
end // class Aclient' interface

class Aclient' implementation
    ...
    methods
        example(x, y : Integer) returns Integer;
            temps
                result1, result2 : Integer;  // will in fact be replaced with a subtype of Integer
            begin
                result1 := server.sendFuture(100, 'faculty' );
                    // spawns a new thread, and continues with next statement
                result2 := server.sendFuture(50, 'faculty' );
                result1.print;      // the print messages are blocked until the result of the
                result2.print;      //   previous invocations have been returned
                return result1 + result2 ;
            end;
    ...
end // class Aclient' implementation
```

The class Future provides the method sendFuture to construct future message invocations. This method does three things; it creates a proxy object and returns it to the client that invoked the sendFuture message. Then it creates a first-class message representation from the receiver and selector that are provided as parameters (for brevity, message arguments are omitted), and fires the constructed message. Finally, the result of the message is handed to the proxy object that was returned to the caller of the future message. This will then unblock all pending requests on the proxy object.

```
class Future interface
    methods
        sendFuture(Any, String, Any ) returns Nil;
    // default filters
end // class Future interface

class Future implementation
    methods
        sendFuture(rec:Any; sel:String) returns Nil
            comment for brevity, we do not deal with arguments;
            temps
                mess : Message;  // creates a first-class representation of a message
                proxy : Proxy;   // creates a proxy object
            begin
                return proxy; // caller resumes execution
                mess.setReceiver(rec);
                mess.setSelector(sel);
                proxy.setReplyTo( mess.fire );  // fire message and assign result to the proxy
            end;
end // class Future implementation
```

The most significant property of the class that implements the proxy object is that it dynamically inherits from the result of the future message: the messages to the result value are all dispatched to an internal var. When the reply of the future becomes available, it is assigned to var. The disp filter makes all the messages of the result object available on the

interface of the proxy. There is a single local method, named setReplyTo, that accepts a reply and assigns it to var.

The synchronisation is defined by the defSync wait filter that overrides the default mutual exclusion[1]. The setReplyTo message is always accepted but all other messages are only allowed when the condition Ready is true. This is the case only after the reply value of the future message invocation has been installed; until this is the case, all requests are blocked.

```
class Proxy interface
    internals
        var : Any;
    conditions
        Ready;    // has the result arrived?
    methods
        setReplyTo(Any) returns Nil;
    inputfilters
        defSync : Wait = { True=>setReplyTo, Ready=>* };  // overrides default
        synchronisation!
        disp : Dispatch = { inner.setReplyTo, var.* };
end // class Proxy interface

class Proxy implementation
    instvars
        replySet : Boolean;
    conditions
        Ready  begin  return replySet;  end;
    initial
        begin replySet:=false; end;
    methods
        setReplyTo(returnValue) returns Nil;
            begin var:=returnValue; replySet:=true; end;
end // class Proxy implementation
```

The important ingredients for defining the modified message passing semantics demonstrated in this subsection are (a) the possibility for constructing first-class message representations and activating these, and (b) the creation of concurrency through the early return mechanism, and (c) the ability to specify suitable synchronisation constraints through wait filters.

3.4.2 An Example of Resource Management: Reader-Writer Synchronisation

Reader-writer synchronisation assumes a partitioning of the operations that access a resource into two kinds: *read* operations that only retrieve information and do not change the state of the resource, and *write* operations that do affect the state of the resource. In order to maintain consistency, write operations have to be performed with mutual exclusion -no other read or write method may be active simultaneously[2]. However, it is possible to

---

1 This is because the object encapsulated by the proxy may provide any type of synchronisation and internal concurrency, this should not be restricted unnecessarily by the proxy object.

2 Note that the read and write operations may manipulate complex data structures, in which case a data structure that is being written can be in an inconsistent state when it is read concurrently.

execute several read operations concurrently, which obviously may increase the throughput of the system.

We present two examples of classes implementing reader-writer synchronisation; the first example is a straightforward implementation, that only enforces mutual exclusion during write operations. For the sake of the example, we simply define a read and a write method for the object, these are assumed to access some encapsulated resource:

```
class RdrWrtr interface
    comment this class implements a combined reader/writer class with reader priority ;
    conditions
        default.Free;
        NoWrtrActive;
    methods
        read returns Any;
        write returns Nil;
    inputfilters
        defSync : Wait = { NoWrtrActive=>read, Free=>write };   // override mutual exclusion
        disp : Dispatch = { inner.* };
end;

class RdrWrtr implementation
    conditions
        NoWrtrActive begin return ^self.activeMethod('write')=0;  end;
    methods
        read returns Any;  begin ...  end ;
        write returns Nil;  begin  ...  end ;
end;
```

The synchronisation is specified by a wait filter with the label defSync in order to override the default mutual exclusion. Mutual exclusion is enforced, however, for write messages, applying the condition Free that is available by default, whereas the read messages are accepted as long as there is no write message executing within the object.

However, this implementation may lead to starvation of write messages. A series of infinitely arriving read messages will all be accepted as long as there are still one or more read messages active. Meanwhile, no write message in the queue of the object can proceed. Thus, the read messages have a priority over the write messages[3]. It is also possible to give priority to write methods by allowing read methods only to execute when there are no write methods active or in the queue. But this approach again might lead to starvation of read methods.

A modified[4] implementation that gives equal priority to read and write messages is defined below. Equal priority means that all messages are served on a first come, first serve (FCFS) basis, where subsequent read operations can execute in parallel. Such an implementation,

---

3 Sometimes the notion of read priority (respectively write priority) is used to designate a situation where an arriving read request has priority over all the write requests currently in the queue. This is not the case here.

4 Extension of the example through inheritance would be equally well possible, although requiring additional condition or method definitions by the RdrWrtr class.

however, cannot be realised by an object with a single message queue, since messages do not have any information regarding their (relative) position in the queue.

The problem that we face is that we cannot distinguish between read messages in the queue before and after the first write message. The read requests that arrived before the first write are allowed to execute (when there is no writer method active in the object) whereas the read messages that arrived after the first write in the queue are not acceptable yet. The adopted solution is illustrated by the following diagram:

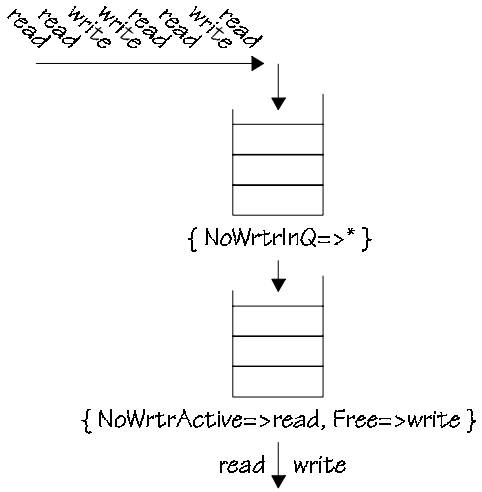


Figure 3.4.1 The multiple queues for the equal priority readers/writers solution

The solution is based on using two messages queues, to separate the 'early' messages from 'late' messages. Messages can pass the first message queue when there is no write message in the second queue; thus there can be at most a single write message in the second queue. Because no more read messages are allowed after the first write message has entered the second queue, the 'late' read messages do not compete with the write message:

```
class RdrWrtrEP interface
    comment "This class allows concurrent read operations, but enforces mutual exclusion
             for write operations. Read and write operations have equal priority: they are
             served on FCFS basis"
    conditions
        default.Free;
        NoWrtrActive;
        NoWrtrInQ;
    methods
        read returns Any;
        write returns Nil;
    inputfilters
        equalSync : Wait = { NoWrtrInQ=>* };  // ensure equal priority
        dummy : Error = { * };  // dummy that separates the wait filters, to create two queues
        defSync : Wait = { NoWrtrActive=>read, Free=>write };     // override mutual
        exclusion
        disp : Dispatch = { inner.* };
end;  // class RdrWrtrEP interface

class RdrWrtrEP implementation
    conditions
        NoWrtrActive begin return ^self.activeMethod('write')=0;  end;
```

```
        NoWrtrInQ begin return ^self.blockedFor(defSync, write)=0;  end;
    methods
        read returns Any;  begin ...  end ;
        write returns Nil;  begin  ...  end ;
end;   // class RdrWrtrEP implementation
```

The class introduces two additional filters, respectively equalSync and dummy. The latter is a rather inelegant tool for separating the two wait filters so that they get their separate message queues[5]. The first message queue is then synchronised by the equalSync wait filter, that imposes the condition NoWrtrInQ on all messages. This condition is expressed using the blockedFor method of the object manager that we previously introduced, supplying the filter label and message selector as an argument. It returns the number of write messages that are in the message queue associated with the defSync wait filter. The synchronisation of the second message queue is the same as for the previous RdrWrtr example.

This example demonstrates that multiple message queues can be used to perform synchronisation that depends on order of message arrival, even though we do not support direct access to the message queue as a data structure. It is important to note that in the case of inheritance or delegation, multiple message queues are created inherently, and this can be exploited for similar purposes. The difficulty with this is that it will often require the subclass to access the object manager of the superclass. The superclass must make this available through conditions or methods.

3.4.3 Inter-Object Synchronisation

As an example of inter-object synchronisation we use the bounded buffer, which functions as an intermediate between two or more clients. We show two extensions of the BoundedBuffer class that are typical representatives leading to certain categories of inheritance anomalies [Matsuoka 93a]. The first example is a composition of multiple synchronisation constraints from different classes, the second example demonstrates history sensitiveness.

Synchronisation Composition

We illustrate the composition of concurrent objects by composing a bounded buffer with a class providing 'locking' features. First the class Locking is defined. It provides two methods, lock and unlock, which respectively 'lock' the object, causing no method except for the unlock to be accepted, and 'unlock' the object, causing all methods to be acceptable again. The status of the object is stored in a boolean instance variable, free.

```
class Locking interface
    methods
        lock returns Nil;
        unlock returns Nil;
```

[5] We have also been experimenting with operations on -sets of- filters, where wait filters sharing a message queue are surrounded by the brackets '<' and '>'. For simplicity we use the implicit merging rule for subsequent wait filters.

```
    conditions
        Unlocked;
    inputfilters
        locksync:Wait={ True=>unlock, Unlocked=>* };
end;

class Locking implementation
    instvars
        free:Boolean;
    conditions
        Unlocked  begin return free; end;
    methods
        lock  begin free:=false; end;
        unlock  begin free:=true; end;
end;
```

The method unlock is always acceptable, independent of the state of the object. The other methods of the object are only allowed when the object is the Unlocked state. Note that, due to the usage of the wild card "*", this synchronisation specification is open-ended in the sense that the specification will still be valid when new methods are added (assuming these have to be blocked in the lock state as well). This open-endedness makes it possible to reuse the synchronisation specification directly in another context.

Class LockingBuffer is a composition of class Locking and class BoundedBuffer that was shown in the previous section. Therefore instances of these two classes are provided as internals. Because the synchronisation that is defined by these two classes needs to be combined, their synchronisation filters are directly used for this class:

```
class LockingBuffer interface
    internals
        buf:BoundedBuffer;
        locker:Locking;
    inputfilters
        locker.lockSync;
        buf.bufferSync;
        disp:Dispatch={ buf.*, locker.* };
end;
```

This realises an AND-condition for the constraints in the subsequent filters (since the constraints imposed by both filters have to be satisfied in order to let the message be accepted). The last filter dispatches messages to the respective internals. Note that their filters have to be passed again, (redundantly) imposing the same synchronisation constraints.

History Sensitiveness

As we argued before, history sensitiveness can be tackled through the provision of a mechanism that can modularly maintain an administration of relevant history aspects. In the composition-filters model, this can be achieved through the application of ACTs. We use the example of the HistoryBuffer that was introduced in section 3.2 to illustrate this. The HistoryBuffer class extends the bounded buffer with a gget method that should not be executed immediately after the execution of a put message.

The HistoryBuffer class first performs synchronisation and registers messages when they are dispatched for execution. This requires that all the messages that pass the meta filter are immediately dispatched and are not blocked by the internal buf anymore. To avoid this, the synchronisation of buf as specified by the bufSync filter, is repeated here:

```
class HistoryBuffer interface
    internals
        buf : BoundedBuffer;
        admin : HistoryACT;  // encapsulated ACT that administers history information
    methods
        gget returns Any;
    conditions
        admin.NoRecentPut;
    inputfilters
        sync : Wait = { NoRecentPut=>gget, True~>gget };
        buf.bufSync;  // or we would register messages in the queue of buf as being executed
        register : Meta = { [*]admin.register };   // all passing messages are registered
        disp : Dispatch={ inner*, buf.* };
end;
```

The ACT object, HistoryACT, receives meta-messages and sets a boolean flag when the message is a put message. The condition NoRecentPut looks at the boolean variable to determine the acceptance of the gget message:

```
class HistoryACT interface
    conditions
        NoRecentPut;
    methods
        register(Message) returns Nil;
    inputfilters
        disp : Dispatch={ inner.* };
end;

class HistoryACT interface
    instvars
        justPut : Boolean;
    conditions
        NoRecentPut begin  return justPut.not;  end;
    methods
        register(meta:Message) returns Nil;
            comment  "updates the instance variable <justPut> and resumes execution of the
                      reified message by firing it.
            begin  justPut := (meta.selector='put'); meta.fire; end;
end;
```

Note that this example does not truly *solve* the inheritance anomaly, but avoids its occurrence. The commonly available techniques for implementing history sensitiveness are not modular, and therefore cause an inheritance anomaly (i.e. not the synchronisation itself, but the code for maintaining history information causes the inheritance anomaly). However, ACTs make it possible to maintain history information in a modular way, integrated in the object-oriented model, and therefore do not cause inheritance anomalies.

3.4.4 Summary

In this section, we made an attempt to demonstrate the expressive power of synchronisation and concurrency control in the composition-filters model through a number of examples in several categories. Additional examples can be found in [Bergmans 92c] and [Bergmans 94].

3.5 Discussion

In this section we briefly discuss the contents of this chapter. The proposed synchronisation scheme is evaluated by comparing it with the requirements that were formulated in section 3.1 and 3.2. Some related proposals that focus on the combination of inheritance and synchronisation constraints are discussed and the chapter is concluded.

3.5.1 Evaluation of our Proposal

We evaluate the synchronisation scheme we proposed in this chapter by comparing its characteristics with the criteria that were defined in 3.1.5:

- *Modular concurrency*: the issue of modular concurrency involves a number of different aspects:
  1. Every object in the composition-filters object model may define its own synchronisation constraints, i.e. all objects are active.
  2. Intra-object concurrency is supported and nested objects may encapsulate threads.
  3. The method implementations of the objects are completely separated from the condition implementations and the wait filter specifications. This achieves synchronisation modularity.
  4. The synchronisation specifications are decomposable at two levels in our proposal: filters can be reused separately and combined into a new synchronisation specification, and the constraints specified by conditions can be reused separately. However, it is not clear whether a different granularity for decomposition would be more appropriate: very fine-grained decomposition can be tedious and ineffective, whereas coarse-grained decomposition may be ineffective as well. The composition-filters model tries to offer a workable compromise.

  Concluding, our approach meets all the basic requirements for modular concurrency.
- *Expression-power*: The mechanism of interface control that is adopted -synchronising messages on the interface of the object- is sufficiently powerful to express most synchronisation problems. In particular because our scheme allows recursive calls to be synchronised as well, thereby enabling synchronisation in the middle of a method body as well. The various examples in section 3.4 have demonstrated the applicability of the synchronisation scheme to a range of different synchronisation problems. The support for synchronisation reuse promotes the definition and application of generic synchronisation policies such as reader-writer synchronisation or tailored message passing synchronisation. Acceptance of a message can be motivated by various aspects of the object, including the state of nested or external objects, the state of the object manager, the properties of the message, and -indirectly- history information. Although other types of information can be imagined, these cover most practical situations. Additional expressive power is achieved by enabling multiple message queues for more complex synchronisation problems.
- *Encapsulation*: The composition-filters model supports strong encapsulation, which is not relaxed for subclasses. Synchronisation specifications are also encapsulated: only the names of conditions and filters are visible on the interface, their respective

implementations remain hidden. An important characteristic is that synchronisation itself is also encapsulated, in the sense that it cannot be modified by inheriting or calling clients of the object. The synchronisation for locally overridden methods can be relaxed, though.

- *Support reuse and composition of concurrent objects*: reusing objects in a different context is supported because concurrency and synchronisation are encapsulated within objects and because each object is active, i.e. it is protected against concurrent access. The separation of synchronisation specifications from the method implementations makes it possible to reuse each of these separately as well. The issue of avoiding the occurrence of inheritance anomalies is discussed below.
- *Efficiency*: The efficiency aspects of the synchronisation mechanism have not been discussed in this chapter, as this is the main topic of chapter 5. We suffice here with the remark that the overhead of the synchronisation scheme lies mainly in the evaluation of conditions. This means that (a) the programmer bears some responsibility in that complex condition implementations should be avoided, (b) in the absence of any activities (i.e. changes of the object manager state), no re-evaluations are required, and (c) inactive objects incur no processing overhead, as they require no re-evaluations.

Several of these criteria have been tested through a number of examples. Some of these were shown in section 3.4, and a number of other examples appear in [Bergmans 92c, 94]. In conclusion, the requirements that were stated before are essentially all met. For a number of aspects, such as the granularity of decomposition, the expression power and efficiency, no absolute claims can be made, but in general we can conclude that these criteria are fulfilled.

We briefly consider the additional favourable properties that were defined in section 3.1.5:

- *Declarative synchronisation specifications*: the definition of wait filters in the interface of the object specifies the synchronisation constraints for the object separately.
- *Data-consistency*: the definition of mutual exclusion as the default behaviour of objects provides a certain level of data-consistency, as the instance variables of an object are protected against concurrent access. Different default protection mechanisms can be defined as well.
- *Model Integration*: An important design criterion was the integration of the synchronisation scheme with the composition-filters model. By introducing filters of type Wait, synchronisation specifications can be added to applications in a modular and consistent manner. They can be freely combined with other application aspects defined by filters such as multiple views, real-time constraints, multiple inheritance and delegation.

The reusability and extensibility characteristics of the synchronisation scheme can be judged by considering the requirements for avoiding inheritance anomalies proposed in section 3.2:

- *Synchronisation modularity*: This requirement is fully met by the composition filters model. Conditions are the mechanism for making the abstract object state concrete. It is possible for multiple conditions to be valid at the same time. The implementation of conditions is fully encapsulated within the implementation part of the object. Conditions can be referred to by other objects through their name, though. Wait filters specify the

mapping from conditions to message acceptance. Both the conditions and the filters may be reused.

- *Synchronisation granularity*: The polymorphic application of synchronisation constraints to messages is possible, where a condition specifies the synchronisation constraint. All possible associations between conditions and messages are realisable: one or more conditions can be combined into a constraint for one or more messages. The composition of synchronisation constraints is possible at two levels: (1) conditions may appear in wait filter specifications in combination with other conditions to form a new synchronisation specification. And (2) complete filters can be reused, where multiple subsequent filters together compose a new synchronisation specification. The semantics for composing filters are restricted though: filters defined by a single object can be combined only with AND semantics[1], in the sense that the constraints specified by all filters must be satisfied.
- *Expressiveness for Synchronisation Conditions*: the expressive power that is available for specifying conditions is quite elaborate, as we discussed above already. This is mainly due to the fact that conditions are message expressions of arbitrary complexity and the provision of an object manager to access the system state.

We conclude that the proposed approach does not severely violate any of the requirements that we stated. For some issues, such as the granularity for decomposition, and the expressiveness of synchronisation conditions, the composition-filters model adopts a specific solution that may not fully exploit all possibilities. Thus, it should be clear that we do not pretend to propose the ultimate solution. The integration with the composition-filters framework is another important property. We do claim that our proposal can at least meet the reusability and extensibility properties of other proposals for concurrent object models.

This chapter only addresses the computation model and its synchronisation scheme; the issue of reasoning about synchronisation specifications is beyond the scope of this thesis. However, in section 5.2 it is described how a set of Wait filters can be translated into Boolean expressions in terms of conditions that can express the acceptance constraints per message. More research is required to investigate reasoning about synchronisation constraints in the presence of multiple sets of Wait filters, and of inheritance and delegation.

3.5.2 Related Work

We discuss some of the recent related work in more detail, several of the other proposals for COOPLs were mentioned in sections 3.1 and 3.2.

ABCL+

The proposal of ABCL+ [Shibayama 91] is based on the language ABCL/1 [Yonezawa 90], but has a number of extensions to enable reuse of synchronisation constraints: firstly, method guards are de-coupled from the methods. This is done by distinguishing *primary* methods from *constraint* methods. The first define application code, whereas the latter define a method guard. The second extension is the introduction of *transition* methods.

---

1 Through composition of objects that each define their own synchronisation constraints, we can select a particular behaviour. This combines filter specifications with OR semantics.

These can modify the synchronisation of an object -as specified by the constraint methods- through dynamic delegation. A become statement in the transition methods specifies a new 'parent'. By defining a number of parent objects, one for each -relevant- state, it is possible to define state abstractions. Special merging rules are defined for inheriting transition methods.

ABCL+ has in fact a two-tiered approach to synchronisation, adopting both method guards and state abstractions that allow to choose a particular set of methods guards. This approach does not suffer from the tight coupling between method code and state abstractions by a clear separation of primary methods and transition methods. The use of method guards ensure that synchronisation can be based on the current state of the object. Some of the history-sensitiveness problems can be addressed by selecting a proper set of method guards after the execution of a method.

ABCL/onAP1000

The synchronisation scheme proposed in [Matsuoka 93b] for ABCL, that has been named after the implementation on the AP1000 multi-computer, aims at solving the common inheritance anomalies while offering an efficient implementation. The most significant characteristic of the approach is the support for multiple synchronisation schemes: this is motivated by the observation that the anomalies occurring in some existing synchronisation schemes can be avoided by other schemes, but these in turn suffer from other anomalies. Providing the user with multiple strategies makes it possible to adopt a specific tool for tackling each particular problem.

Two synchronisation schemes are proposed: one that is based on *transition specifications*, which is similar to the approach of behavioural abstractions. The common problems with behavioural abstractions are avoided by associating guards with them, and by defining a number of *transition types*. The latter can have particular semantics such as saving and restoring the current message accept set. The second synchronisation scheme is the so-called *synchronizer*. This is essentially a method guard, but a single guard can be associated with multiple methods. Guards can be expressed in terms of the instance variables of the object, message arguments and the acceptance state of methods (i.e. whether a method or method set is enabled or disabled).

The approach of supporting multiple synchronisation schemes has some drawbacks, however: the developer must understand several synchronisation schemes, and choose one that is deemed appropriate. When multiple schemes are required for a class and its subclasses, the interface of all schemes must be maintained[2].

The ABCL computation model does not support intra-object concurrency, which is considered to be important for reusability as well [Nierstrasz 93a]. This restriction allows for more efficient implementations of the synchronisation schemes, however. A general solution for dealing with history information is not available but some specific transition types can solve certain forms of the history sensitiveness inheritance anomaly. Important

[2] It is not fully clear what the effect of multiple schemes is on the efficiency of the synchronisation scheme, this is likely to increase the amount of overhead, though.

characteristics of the approach are the consistent use of method sets for both synchronisation schemes, and the separation of transition specifications from method implementations. These de-couple methods from synchronisation conditions, thereby avoiding the occurrence of anomalies.

Proposal by Frølund

In [Frølund 92] a generic synchronisation mechanism is proposed to support the reuse of synchronisation constraints. One of the starting-points is that synchronisation constraints should be incrementally more restrictive in subclasses. This conforms to our opinion that relaxing the constraints of inherited methods in subclasses breaks encapsulation. However, the paper does mention a mechanism called *iron-curtain*, which enables selective inheritance of restrictions. This mechanism is not worked out in [Frølund 92], but should be applied with care to avoid breaking of encapsulation after all.

The synchronisation constraints must be defined in the form

<guard condition> **prevents** <message pattern>

where the guard condition is a message expression resulting in a boolean and the message pattern selects invocations based on their names and parameter list. The patterns allow the conditions to be applied polymorphically to several messages, although this cannot be specified incrementally (i.e. we cannot associate a new message with a specific restriction in a subclass). The paper mentions that patterns could be made first-class, as to allow them to be named and composed.

A problem with the approach is that conditions are not named, and can thus not be reused. This is partly addressed by the *disabled* construct that indicates the current acceptance state of a particular message. This does allow for composition of restrictions, but depends on the names of messages (essentially breaking encapsulation, since the synchronisation constraints of a particular message are an implementation issue). The other problem is that the *disabled* operation is not a reference to a known restriction, but to the combination of all restrictions for a specific method. As a result, one can refer to the synchronisation of another method, but not to a specific constraint, e.g. as defined in a superclass.

For example, consider adding a method unlockable_get in the locking buffer example. This method has the same constraints as the unlocked get. The expression *disabled(get)* will however depend on the locking state of the buffer as well, there is no way to reuse the synchronisation constraint that were defined for the get only. It seems to be inappropriate to define synchronisation constraints for one method in terms of the synchronisation of another method.

One other property of the proposal is a construct *all-except*, which defines constraints for all methods, with the option of excluding specific methods. This is comparable to the use of the wild card in filter specifications. However, in our understanding, this constraint is inherited and thus applies to the methods defined in subclasses as well. This seems to be a pessimistic approach, as some constraints may not apply to newly introduced methods in subclasses, but they cannot be redefined (except through the iron-curtain mechanism).

3.5.3 Conclusion

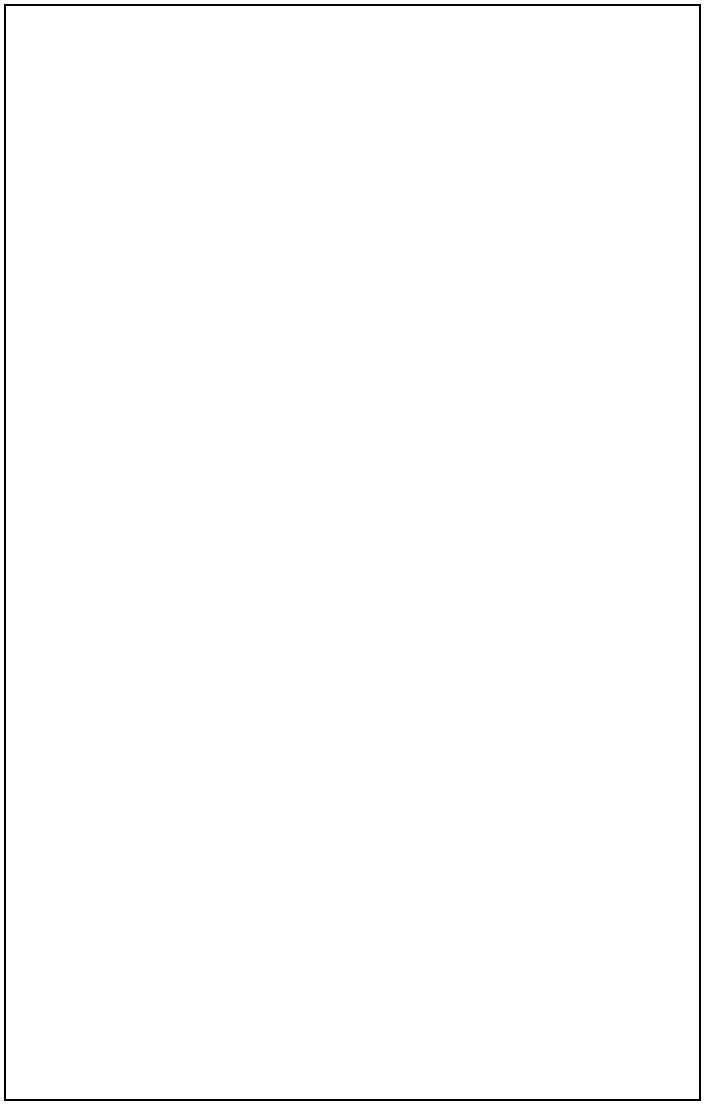
This chapter makes two main contributions: the first contribution is the analysis of the origins of inheritance anomalies, the second is the synchronisation scheme we propose for the composition-filters model.

In section 3.2 a generic framework for synchronisation schemes is defined. Most synchronisation schemes that perform synchronisation of received messages at the interface of the object with object-level synchronisation fit within the framework. The origins of inheritance anomalies are then explained using this framework. As a result, a classification of the origins of inheritance anomalies is given. These are used to derive a number of requirements on synchronisation schemes to avoid the inheritance anomalies.

The synchronisation scheme we propose for the composition-filters model is based on a new type of filter, the wait filter. This filter can be used to specify synchronisation constraints on messages and is fully integrated within the framework offered by the composition-filters model. Concurrency is created through the transparent mechanism of non-blocking RPC message passing semantics. We show that the model can solve a range of synchronisation problems, can avoid all categories of inheritance anomalies to a certain extent, and adheres to all of the criteria for concurrent object-oriented programming languages that were defined in section 3.1.

We stress that the proposed synchronisation scheme is both a powerful synchronisation scheme with good reuse and extensibility properties, and it is smoothly integrated within the composition-filters model.

Chapter 4



Analysis and Design of Concurrent Objects

4.1 Introduction

4.1.1 Our Goal.

This chapter describes a method for the development of concurrent, object-oriented programs. The method focuses on two primary aspects. The first aspect is to guide the analysis and design process through method steps that give hints, guidelines and suggestions. The second aspect is a notation that supports this aspect. Apart from diagrams for describing fairly conventional aspects of objects such as parts, inheritance, method interfaces etcetera, we highlight the so-called *state composition diagrams* (SCDs). These describe the synchronisation of messages on the interface of the objects in an intuitive, yet precise way.

The method we present is based on the premise that we do not address the actual behaviour of objects (i.e. the sequence of events), how to specify it or how to implement it. Instead we focus on the *constraints* that must be satisfied in order to guarantee the consistency of objects, or the system they are part of. We consider a (sub-)system to consist of a set of objects, with a set of processes that execute the methods of objects. In principle, there is no restriction on the number of processes that can be active within an object, or executing a single method. The number of processes may vary over the system life-time, and so may the number of objects that participate within that system.

The problems involved in constructing a complete -static- description of such a system are considerable. A global system description is not appropriate for two reasons: the first is that during the system life-cycle certain parts of the system are likely to be modified or extended, which affects a global description. The second reason is that objects may be created or destroyed at run-time, which makes it difficult to give a global description of the synchronisation. Thus we modularise the description of the system by making objects the granularity of concurrency control. Open-endedness of the object specifications is one of the primary issues.

Our approach is based on the following two statements:

- The description of the dynamic aspects must be fine-grained rather than global. The object level is an obvious choice although for some situations not fine-grained enough[1].
- We describe the minimal constraints that are to be satisfied for ensuring consistency; this leaves open an infinite number of event sequences. The possible sequences are then again restricted by the implementation of the methods of objects (which can be considered as fixed event sequences) and the configuration of objects.

Thus, rather than most other approaches in this area, which focus on fixing the sequence of events as much as possible (they concentrate on specifying the causality relations between events), we want to do this as little as possible. By specifying only the necessary constraints for each component in the system, the components are less dependent of a specific context.

---

1 Cf. the origins of inheritance anomalies.

4.1.2 Related Work

Two categories of related work are distinguished here. The first is methodological support for the dynamic aspects, the second is the notation and its underlying model for representing dynamic aspects.

In the area of methodological support for the development of object-oriented concurrent software there is little related work, to our knowledge; we did not encounter methodological support and notations for developing the synchronisation of objects. The closest related work is done by Denis Caromel; in [Caromel 93] a method is outlined that is based on the parallel language Eiffel// [Caromel 90], we will refer to this method as the *Eiffel// method*. Other research addressing methodological support for a concurrent system appears in the real-time research area, for instance in [Shlaer 92] and [Buhr 92].

Most object-oriented software development methods do address the issue of modelling the causality relations between events. For example, OMT [Rumbaugh 91], OOD [Booch 90], OOA/OOD [Coad 91a,b] and Objectory [Jacobson 92] all address this by constructing a model based on finite state machines or state charts (these are discussed later in this subsection). A common problem with these approaches is that the state models are weakly, or not at all integrated with the inheritance mechanism and often do not support concurrency. In addition the realisation of the state models in the implementation phase is either undefined, or not properly integrated with the application objects. Most methods avoid the issues of concurrency and synchronisation.

The Eiffel// Method

This method is based on the computation model of Eiffel//, as described in [Caromel 90], and inevitably this underlying model affects the method itself. The properties of Eiffel// have been discussed in section 3.1 and 2.3. The method can be briefly described by summing up the method steps:

1. *Sequential design and programming*: here the conventional object-oriented techniques are adopted.
2. *Process identification*: the method associates processes descriptions (*live* routines) with classes. In this step for each of the sequential classes it is considered whether they should be provided with such an initial activity. Then all the objects in the system that are possibly shared by two or more process classes must necessarily be turned into process classes themselves, as this is the only means for synchronising requests.
3. *Process programming*: The most important activity of this step is the (re-) definition of the *live* routines that determine process control and synchronisation. Note that any change to these in a subclass requires the complete redefinition of this routine.
4. *Adaptation to constraints*: This step is primarily intended to tailor the system to meet real-time specifications. It is concerned with the mapping of processes (objects) to processors and the definition of new processes for buffering, distribution of activities, or different priorities. This is a typical iterative step to tune the system.

The Eiffel// method does not incorporate any support, notational or otherwise, for the derivation of synchronisation constraints.

Object Life-cycles

This method is actually described in two books, respectively "Object-Oriented Systems Analysis: Modeling the World in Data" [Shlaer 88], and "Object Life-cycles: Modeling the World in States" [Shlaer 92]. The first focuses on static system descriptions, whereas the latter emphasises the dynamic issues of system modelling. The most important components of the method (we ignore subsystems and domains, that are primarily intended for managing the software development process) are *information models*, *state models*, and *process models*.

Information models are similar to entity-relationship diagrams [Chen 76], and give a static system description. State models describe the life-cycle of individual objects and relationships. The state model is a *Mealy* [Hopcroft 79] finite state machine (discussed later in this subsection); with each state an action is associated that is performed when the state is entered due to the occurrence of some event. The actions may generate new events, that may also be destined at other objects. These interactions between objects are described in an *object communication model*. The actions themselves are worked out into more detail in an *action data flow diagram* (i.e. a process model). The resulting inter-object data access is described by the *object access model*, which is a process model as well.

The state models for objects describe the causal relationships between actions, and are thus not concerned with concurrency and synchronisation issues. However, the state models can also be applied to relationships between objects (relationship objects). To handle competing events, these objects can be turned into monitor-like constructs (*assigners*) that serialise events. State models are described in a graphical notation or in equivalent state transition tables.

The method also allows for expressing the creation and forking of threads but provides no means for specifying more complex synchronisation specifications other than through the *assigner* described above. In the design phase, the method identifies three types of objects:

Passive classes are those that do not have a state model associated with them, and are mapped to a conventional sequential class. *Active classes* have an associated state model which is implemented by inheriting from an abstract class *Active_Instance*. This class provides an interface to a finite-state machine simulator object. Through proper initialisation, for each received event, the finite state machine simulator determines the appropriate new state. *Assigner classes* are similar to active classes but they consist of nothing but a state model that is implemented as just described for active classes. The method does not address the issues in implementing and synchronising concurrent threads.

In summary, although the method gives a rather complete and pragmatic approach to the analysis of systems, it cannot model the complex synchronisation constraints that we are interested in, and provides only partial support for the realisation of state models.

Models and Notations for Describing Dynamic Behaviour

We describe a number of approaches to the modelling of dynamic object behaviour (in the sense of causal relations between actions). Practically all approaches are based on some variation of finite-state machines. A finite state machine describes a number of states a system can be in, and transitions between those states. A transition arrow from state A to

state B describes the *event* α that causes the transition from state A to state B. Two variations exist for adding actions to the finite-state machines; the *Moore* respectively the *Mealy* model. The *Moore* model [Moore 56] attaches actions to the entrance into a new state, whereas the *Mealy* model [Hopcroft 79] associates actions to the transition. Note that a finite state machine *prescribes* a computation; it fixes the causal relations between successive events. This is not the same as a description of synchronisation constraints.

We will now briefly describe the formalism of *state charts*, which comprises a number of extensions to the basic finite-state machine formalism. Most object-oriented methods and techniques (e.g. ObjectCharts [Coleman 92], OMT [Rumbaugh 91] and OSA [Embley 92]) adopt state charts as the basic mechanism for describing dynamic object behaviour even though the original work on state charts had no connection with the object-oriented approach. The other approaches we will describe are within an object-oriented, or at least object-based, context.

State Charts

The work of Harel [Harel 87] extends the basic notion of a finite-state machine to the so-called *state charts*. One of the problems with finite-state machines is that they do not scale up well; the addition to a diagram of, say, 2 new states that are orthogonal to the existing *n* states, requires the definition of $2 \times n$ new states. State charts try to solve such complexity problems by defining two operations (expressed graphically) for combining state charts: orthogonal combination and refinement.

State charts feature three important extensions to finite state machines:

1. *Depth*: This is also referred to as hierarchy or refinement. It means that for each state a number of substates can be defined along with the transitions between the substates. Depth provides a means for managing complexity by constructing a hierarchy of nested state charts.
2. *Orthogonality*: This means that state charts can be combined such that the system is assumed to be in two (or more) states at the same time. The transitions in orthogonal state charts can be completely independent. This is another tool for managing complexity by building a system description as a combination of state charts.
3. *Broadcast*: The actions associated with transitions are broadcast throughout the entire system. This is also a means for communication between otherwise independent orthogonal state charts. Note that this property contrasts with the approach in object-oriented systems, where communication between objects is purely based on peer-to-peer message passing.

State charts are a powerful and expressive yet intuitive approach to modelling the causal relationships between events in a system. This has made it the premier model and notation for a wide range of software development methods.

Objectcharts

In [Bear 90] and [Coleman 92], a notation is presented for specifying the behaviour of objects, called *Objectcharts*. Objectcharts are based on state charts from Harel but do not adopt the broadcast mechanism, and integrate transitions with the message invocations (request-reply model) of the object-oriented model. Objectcharts provides a graphical

notation and adds more detail to this with a formal specification of the transitions (start-state, ending state, firing condition, and post condition) and invariant specifications (specifying states).

An interesting aspect is that the model allows for making statements about the behaviour of a system of interacting objects. However, this requires the restriction that the number of objects is static; no objects can be added or removed at run-time. Other restrictions are that during a method execution (service request and execution in ObjectChart terms) no messages can be sent to other objects, and that no inheritance mechanism for reusing objectcharts is provided (inheritance subtyping relations can be verified afterwards though). In [Coleman 92] the steps of a method are outlined, without further details on methodological support.

Object-oriented Systems Analyis

The object-oriented analysis method OSA [Embley 92] adopts a notation called *state nets* for modelling the dynamic behaviour of systems. The *state nets* are an extension of state-transition diagrams. Its major features are:

- A transition consists of a condition and an action. The transition will only take place when the condition is satisfied. During the transition the associated action is performed, which may take some time (i.e. a transition is not considered to be atomic).
- The model supports specialisation of state nets in subclasses:
  - A single transition can be specialised, e.g. with a more specific condition or action.
  - New transitions and states can be added in a subclass.
- A transition may have multiple in- and out-arrows, i.e. it can have multiple start and ending states. This allows for expressing forms of synchronisation, for example a transition may only occur when the object is in all start states. It also supports (intra-object) concurrency. For instance after a transition the object is in multiple subsequent states, from each of which new transitions may occur independently.
- The model supports exceptions and real-time constraints (i.e. deadlines and time frames on transitions).

The OSA model offers considerable modelling power but does not address the topic of how to map the analysis model to a design model and eventually to an implementation.

ObjChart

ObjChart [Gangopadhyay 93] is a visual formalism to specify objects and their reactive behaviour. A system is specified as a collection of asynchronously communicating objects arranged in a part-of hierarchy (inheritance is not supported). Each object in the system is specified by a finite state machine. This implements its reactive behaviour in reply to received messages. Messages received by an object are events that trigger state transitions (provided the associated condition is true). With each transition, an action is associated. An action consists of sending one or more asynchronous messages to either itself, its components (*siblings*) or one of its ports. Relations between objects can be expressed by either stating functional invariants over their respective attributes, or by a finite state machine specifying behavioural dependencies between ports of the objects.

One of the most interesting properties of ObjChart is that it is an executable model. Thus, every specification is immediately testable, and the constructed models are executable. It is shown that by object composition only a single state machine is required per object and no combinatorial explosion of states will occur. Since the actions of a state transition are always local (i.e. may affect only the object itself, its components or its ports), semantic analysis can be performed efficiently, compared to the state charts approach.

One of the problems of the ObjChart approach for our purposes is that it makes objects the unit for state-transition diagrams, and combination of these diagrams (in order to prohibit the combinatorial explosion of states) can only be achieved through object composition. While this results in a model that is elegant by itself, we feel this may conflict with modelling issues: the granularity of the finite state machine may be finer than the granularity required for data abstraction. In the ObjChart approach the state-transition diagrams (including the extensions) covers basically all aspects required for defining objects, including the specifications of methods. We are more interested in an approach that highlights the dynamic aspects of objects, but does not compete with other tools and methods steps used in software development.

4.1.3 Requirements

To make our objectives in defining a method for developing concurrent objects explicit, we define a number of properties that the method should preferably accommodate. These properties are separated again into two distinguished sets: the first is a set of requirements for the method itself. The second is a set of requirements on the notation that is used by the method.

Requirements for the method:

The following list presents the most important criteria for judging a method:

1. *Support for modelling and solving concurrency issues:* This is our main interest. The method we describe is mainly just a framework in which the methodological support for concurrency and synchronisation issues (we will refer to this as the *dynamics* or *dynamic aspects*[2] of the application) is embedded. Note however, that the emphasis on concurrency and synchronisation issues will affect even the method steps that are seemingly unrelated. For example, the decisions with respect to the (largely static) structural relations between objects may be affected by issues such as the placement of synchronisation boundaries, inter-object and intra-object concurrency, and synchronisation reuse.
2. *Integration with OO principles and the Composition-Filters model:* Our aim is to provide a method that supports the development of object-oriented applications with the composition-filters model. The integration takes two forms: first of all, we want to integrate the dynamic aspects with the conventional sequential object-oriented concepts. This means that the dynamic aspects must be integrated within objects such that

[2] This should not be confused with what is sometimes also referred to as the dynamic aspects of an application: the flow of control due to message sending and control structures, as can be found in sequential programs.

concepts as message sending, encapsulation, polymorphism and inheritance are still meaningful for the dynamic objects. Secondly, we strive to apply these object-oriented principles to the dynamic aspects themselves. This should be beneficial for the reusability and extensibility of the dynamic aspects as well.

3. *Support open-endedness and extension:* As software systems continue to evolve during their life-time, open-endedness is an essential property for software. The dynamic aspects of objects are no exception to this. Therefore the specification of objects must be open-ended and extensible. This first of all requires a model that allows this, but the method must support this as well. A method must guide and steer the developer to make design decisions that improve the open-endedness and extensibility of the developed application objects.
4. *Support Reuse:* Reuse of software is one of the much-quoted promises of object-oriented programming. Reusing existing objects should make the software development process less tedious and time-consuming, improve its manageability, and produce more robust software. The dynamic aspects of systems are particularly hard to analyse, specify and implement, and thus will profit accordingly from reusing existing components. The software development method is expected to provide techniques for constructing reusable objects on one hand, and to provide techniques and guidelines to achieve the actual reuse on the other hand.
5. *Reduce the implementation gap:* The conventional software development methods often suffer from the problem that the specification model that is the product of the analysis and design phases stands too far from the implementation model. This is either because the specification is too high-level, or because the specification model is essentially different from the implementation model. This problem is referred to as the *implementation gap*. The object-oriented approach to software development reduces this problem by taking the same underlying concepts for expressing analysis, design and implementation models. However, this only applies to the basic and static concepts.

   We try to further bridge the gap between specification and implementation with automated techniques. Therefore we require the specification model to be transformable to an implementation model[3].
6. *Support iterative development:* Our approach to analysis and design is model-driven, rather than method-driven; we emphasise the construction of a model of the system, starting with a simple, high-level picture, and gradually developing a more detailed description. This is a process where the steps in the methods are performed in varying order, often repeated several times on the same or different parts of the model. The methodology steps and notations should be defined with such chaotic but more realistic advancement in mind.
7. *Intuitiveness:* A software development method is to be applied (largely) by humans. Thus, it should try to cope with the abilities and limitations of human beings. This

[3] Note that this is not the same as the generation of an *executable model*. This is because the transformed specification may still be an incomplete object specification, requiring a number of additional implementation details to be added before being executable.

influences the way the development process is managed and the techniques and notations.

Requirements for the Notation

Designing a graphical design notation is not a purely technical activity: the notation is to be used within the human activity of software development. This requires the notation to be intuitive, aesthetic (although this is a very subjective notion), and most important, convenient. It is hard to verify to which degree these issues can be satisfied, however[4].

In [Ackroyd 91] the following guidelines for notations were proposed:

1. A notation must be composed of a small number of simple basic symbols.
2. The notation must be easy to draw by hand or with automated tools.
3. The visual impact of the arrangement of the basic symbols should connote the semantics of the situation they represent (i.e. it must be intuitive).
4. The notation must be unambiguous.
5. It should be possible to apply the elements in the notation polymorphically to other notational elements.
6. The notation must be expressive (e.g. compared to source code).

We would like to add the following two requirements:

7. It should be possible to keep multiple views or diagrams dealing with the same element consistent, or at least to verify that they are not conflicting.
8. The notation should have well-defined semantics. Preferably, the specifications expressed in the notation must be suitable to be mapped to an implementation model.

Further, we feel that guideline 2. should be stated stronger in the sense that a notation should preferably be suitable to both drawing by hand and by automated tools. At the end of this chapter, we will consider to what extent the requirements for the method and notation are satisfied by the method and notation that are introduced in this chapter.

4.1.4 The Approach

Our approach to the analysis and design of concurrent object-oriented systems can be summarised as follows: we want to provide a software development method that addresses all steps in the development while focusing on the determination of synchronisation constraints. Synchronisation takes place at the object level, and must be suitable for an implementation in the composition-filters model. As a part of the method we require a graphical notation that describes the synchronisation of messages at the boundary of objects. We want to guarantee that the transition from the notation to an implementation is unambiguous, straightforward, and preferably suitable to be automated.

On the Use of Annotations

Because our primary interest lies within the area of synchronisation, the state composition diagrams basically focus on the states and the acceptance of events. Events can be either

[4] This requires empirical experiments in which the application of different notations is compared. This is beyond the scope of our research.

received or sent by an object. In addition, as a notational convenience, we provide a notation for indicating causality between events. These are basically *design annotations*. However, when worked out into complete detail, they could function as a complete specification of the methods of the object! We do not emphasise this approach to dealing with causality relations for various reasons:

a) This can only be used in very fine-grained (small, uncomplicated) objects, for otherwise the diagrams become too large and complex, and become hard to manage.
b) We are primarily concerned with synchronisation and concurrency control. Thus, we do not claim to present a complete method including full support for the definition of method bodies[5].
c) In general the actions of a method may be numerous, but only few of these are interesting for the dynamic aspects of the object. Enforcing the entire method body to be specified as a substep of defining the dynamics of objects seems to be inappropriate.

The reason we *do* allow such annotations is that they can be very useful for the designer to form a picture of the system (object) under development. They may complete the life-cycle of the object. Rather than rigorously ruling out such vital information, we feel it is more appropriate to include it in the notation, but we can only give hints as to how they will appear in the eventual implementation of the object. Note that in principle it *is* possible for the developer to specify the complete method implementations as the actions that are associated with an event (although this may be inconvenient when an event corresponds to multiple arrows in the diagram).

Other methods and techniques either do not speak about the realisation of the causality relations expressed by state transition diagrams, or they leave it to the software engineer to do an ad-hoc implementation that realises the specification of the state transition diagram (e.g. [Booch 90]), or they support this by constructing a finite state machine simulation that is fed with the states and transitions (e.g. in [Shlaer 92] and [Rumbaugh 93]). However, the latter two situations result in the construction of programs that are structured around finite state machines rather than objects. In our opinion this collides with other activities of object-oriented software development.

Why a State-Transition Diagram Approach?

The State Composition Diagrams we will introduce aim at providing an intuitive, visual formalism for describing the synchronisation constraints of objects. Although there are a large number of formalisms for describing such dynamic system aspects, few come with attractive visual representations. State-transition diagrams are one of the main exceptions.

We adopted the state-transition paradigm since it integrates well with the object-oriented computation model: states can be viewed as an abstraction of the status of a system, or of a particular object. Transitions provide an intuitive match for message invocations, as a message invocation will in general lead to a state change. As we are interested in

5 We consider this important, as a number of techniques for object-oriented software development are available that address the specification of method bodies. We do not want to compete with or overrule such techniques.

investigating the relation between the status of objects and -synchronisation of- messages, state transition diagrams model exactly the components that are relevant.

The choice of adopting a mechanism that resembles state-transition diagrams is fairly common: a large number of methods and techniques adopt variations of state-transition diagrams. It is the model of choice for both conventional methodologies [Ward 85], [Yourdon 89] and, as we discussed before, object-oriented methodologies (Shlaer & Mellor [Shlaer 88, 92], OOD [Booch 90], OMT [Rumbaugh 91], Objectory [Jacobson 92] and OSA [Embley 92]). In these methodologies, the state-transition diagrams are primarily used to obtain an additional perspective of the system; the dynamic system behaviour. However, these methods often do not take the information that is specified by state transition diagrams explicitly into account in the subsequent design and implementation phases. In the USE method [Wasserman 85] extended state transition diagrams are used for the specification of human-computer interaction.

State transition models are also frequently adopted by techniques that focus on dynamic system modelling, such as State Charts [Harel 87], ObjectCharts [Coleman 92], ObjChart [Gangopadhyay 93] and State Nets [Embley 92]. The difference with the general methodologies discussed in the previous paragraph is that these techniques pay more attention to the semantics of the resulting diagrams, and try to exploit these. For example, in ObjChart, a completed diagram can serve as an executable specification.

This touches on an important issue; the mere description of dynamic behaviour is no guarantee that this specification is actually taken into consideration during later methodological steps. Therefore, the methodology should explicitly address the incorporation of the dynamic aspects into the overall system architecture.

Another issue that we would like to emphasise is that the analysis and design of the dynamic aspects of objects is just one of the elements of the software development process, albeit one that we focus on in this. Thus it is important to us that the results are applicable in a wider context, and can be made a part of a complete software development methodology. The ultimate goal of this methodology is to come up with a well-balanced analysis, design and implementation that meets all criteria and properties, such as maintainability, reliability, extensibility, etcetera.

4.1.5 The Organisation of this Chapter

This chapter is organised as follows: in section 4.2 the notation for the static aspects of objects is introduced. We term this notation *object diagrams*. This consists actually of several types of diagrams. *Basic object diagrams* describe the properties of bare stand-alone objects. In *object structure diagrams* the structural relations with other objects are described, and *object interaction diagrams* describe which message interactions with other objects occur.

Section 4.3 then describes the notation for the specification of synchronisation constraints: *state composition diagrams*. Apart from the notational aspects the semantics of the diagrams are discussed.

The method is described in section 4.4. It consists of the presentation of eight method steps. Before describing these, the running example that is used to illustrate some aspects of the

method steps is introduced. This is the well-known synchronisation problem of the dining philosophers. The section concludes with a description of the software development process.

In the next section, 4.5, the translation process from the notation to the composition filters model is described. This presentation uses the translation of the dining philosopher example to demonstrate the mechanism, followed by a discussion of the translation for each of the components in the object diagrams (which include the state composition diagrams). Section 4.6 concludes this chapter with an evaluation of the presented method and notations.

4.2 Object Diagrams

The software development method that is introduced in this chapter is supported by a number of graphical diagrams. Each of these diagrams is intended to contribute to the development of a specific aspect of the application. But although each diagram highlights a particular issue, they all have in common that they deal with the notion of objects. Hence, all diagrams are based on the same graphical notation for representing the essential aspects of objects. A diagram only containing these core aspects is called a *Basic Object Diagram* (BOD).

4.2.1 Introduction

An important property of the notations presented here is that they are completely integrated. This has two major benefits: firstly, it presents common aspects in a consistent manner, and thereby reduces the learning curve. Secondly, the integration of the notations enables an incremental and iterative development approach. This is of vital importance, especially for computer-supported tools. It poses some restrictions, though: it requires that every change in any of the diagram types can be completely and consistently mapped to all other diagram types. This is achieved if for every notational element in each of the diagram types one of the following holds:

- The element is either completely orthogonal (with respect to the object semantics it represents) to all other notational elements in all other diagram types.
- The element is common to all diagram types.
- There is a one-to-one mapping to another notational element or to a set of other notational elements.

Note that it is important that in the latter case a transformation function can be defined in both directions (i.e. the transformation function is a bijection).

The following figure defines the specialisation relations between the different types of diagrams:

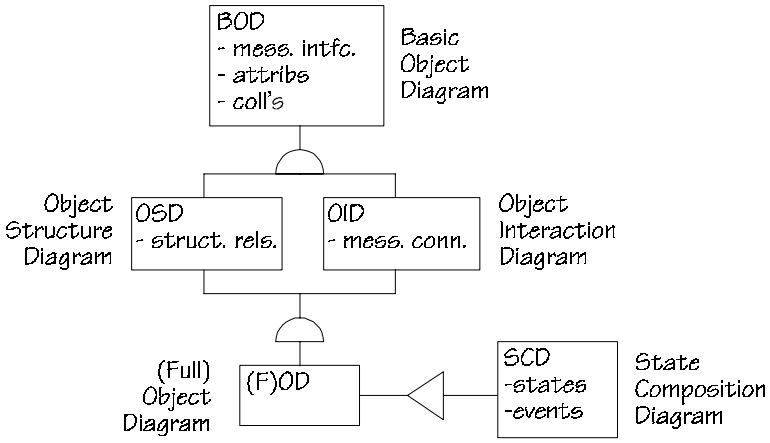


Figure 4.2.1 An (OO) description showing how the various diagrams relate to each other.

As is shown in this (meta-level) figure, the *Basic Object Diagram* defines the common characteristics of all diagrams (therefore it is shown as the common superclass). The most

important characteristics are specified as attributes, preceded by a dash. These include the definition of an interface (i.e. the incoming and outgoing messages), of nested objects, and of object collections. Each of the subclasses in the figure adds a particular segment of information to these characteristics.

> The various diagrams can also be interpreted as different *views* on the same object(s). Thus, the *(Full) Object Diagram* in the figure above represents the diagrams that contain complete object descriptions, including all details. Towards the end of the development cycle all aspects of the object diagrams should become fully defined. The reasons for introducing distinct diagrams along the way are two-fold: firstly, from a software engineering perspective the reduced complexity and finer granularity provides for better ways of managing the development. Secondly, the use of specialised diagrams enables the method -and thereby the software developer- to focus on one particular aspect at a time. Besides, it allows for the notation to be extended modularly with additional notations that can deal with other aspects.

The *Object Structure Diagrams* (OSDs) deal with the (object-oriented) structural relations between objects, the *Object Interaction Diagrams* (OIDs) deal with the message interactions between objects. In this section, these two diagram types and the BODs are discussed. Section 4.3 discusses the *State Composition Diagrams* (SCDs), which complete the *Full Object Diagrams*.

4.2.2 Basic Object Diagrams (BODs)

Objects and Subsystems

We will explain the various diagrams with an example of a company that produces and sells products. This activity is essentially a producer-consumer process. The machines in the production department (the producer) and the sales department (the consumer) are separated by the warehouse (a bounded buffer, named storage). The subsystem representing the company is represented as follows:

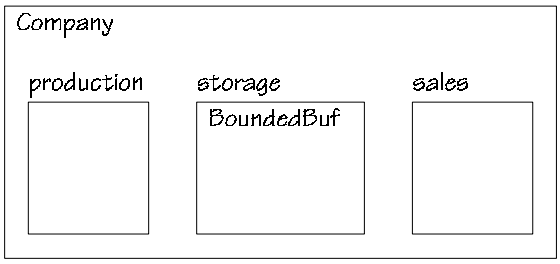


Figure 4.2.2 The company subsystem.

In object diagrams, objects are drawn as rectangles. Because subsystems are fully-fledged objects[1], no distinction is made between the two. Objects within a subsystem are considered to be *parts* of the encapsulating subsystem (object).

---

[1] The reasons for using objects to represent subsystems are two-fold: firstly, because different semantics for subsystems and objects may cause problems during the software development cycle, as described in [Aksit 92b]. Secondly, contrary to the conventional subsystem semantics, objects enforce encapsulation. This significantly improves the manageability and maintainability of complex systems. The inconvenience that may be caused by these semantics can be largely

With each object a label can be associated. It must be placed in the top left corner, just outside the rectangle representing the object. This label identifies the object. At the top, just within the rectangle representing the object, the class name[2] of the object is specified. Thus, in the figure above, there is a subsystem of class Company, which contains three objects, respectively labelled production, storage and sales. Only for the storage object its class name (BoundedBuf) is specified.

The distinction between classes and instances is somewhat blurred in object diagrams. This is because object diagrams are primarily intended to present the dynamic aspects of an application, and therefore focus on the configuration of instances at run-time. Ultimately, all these configurations are collected in the class description of an object. Therefore, we basically ignore the distinction between classes and instances. For example, inheritance relationships, which are by definition relations between classes, are supported by object diagrams. In addition, this dual nature of objects allows for the specification of relations between classes and instances, for example delegation relations.

Note that two or more objects representing instances of the same class may appear in different configurations: in such cases, the combination of all these appearances make up the class description. In other words, the specification of a class is defined by the sum of all appearances of instances of that class (i.e. in all possible configurations).

Collections of Objects

Since an object diagram aims at describing the configuration of objects in an actual -instantiated- application, it can be important to distinguish single, unique objects from multiple objects (in a certain context). This is achieved by defining collections of objects (cf. [Gangopadhyay 93]). In a collection, objects can be accessed through an index, and can be added to or removed from the collection. For instance, in our company example, we would like to model multiple machines within the company, while there still is a single storage and sales department. Figure 4.2.3 shows the notation for representing this in object diagrams:

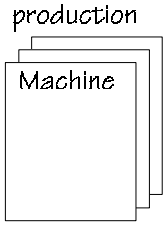


Figure 4.2.3 Notation for specifying a collection of objects.

The collection is tagged with a label (production) that designates the collection of objects, rather than a single instance[3]. The type of the elements in the collection (Machine) can be

solved through appropriate scope rules (see for example section 2.5 for the scope rules defined by composition-filters computation model).

2 When the object at hand is a *reference* to the specified class, rather than a *definition* of a new class, the class name should be interpreted as a *type* specification: the name of the class designates a set of classes that are type-compatible.

3 Actually, the collection is a distinct object itself, that encapsulates a varying number of objects. Accessing, adding and removing objects is done by invoking methods of the collection object.

specified inside the rectangle. Instances may be added to or removed from the collection at run-time (depending on the requirements of the specific application). This is important as it is the sole mechanism for dynamically adding and deleting objects in a system that the method supports.

Note that the notion of multiplicity is relative to the context of the collection. Multiplicity is specified explicitly because it influences the interaction patterns with other objects in the same subsystem. For instance, in the example system, multiple machines may interact with a single storage object. This is important information when designing the storage object, especially with respect to its synchronisation. Note that, if no multiplicity is specified for an object, this does not necessarily mean that there is only a single instance of that object in the whole application.

Parts and Attributes

Basic object diagrams reveal the internal structure of objects. This structure is encapsulated within the object, and it could be argued that this should remain hidden at this level of abstraction. The internal structure plays a major role in characterising the object, though. As the internal structure is essential for nearly all other types of diagram and it is of major importance for the developer, it is incorporated into basic object diagrams.

The internal structure that is revealed consists solely of nested objects. These appear in two notational forms. The first form is graphical, and simply visualises object nesting. The term *parts* is used to designate this notation for nested objects. The parts are drawn within the rectangle that defines the encapsulating object. The notation for drawing parts does not differ from the one for drawing independent objects.

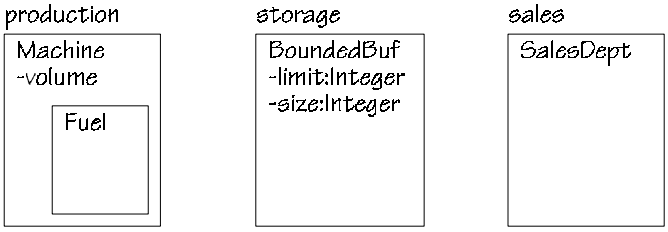


Figure 4.2.4 Objects, nesting and attributes

The second form for specifying nested objects is as a textual list. This requires less space in a drawing, which is especially useful when drawing diagrams by hand. The nested objects that are specified this way, are referred to as *attributes*[4]. The list of attributes is specified just below the name of the class, for instance:

BoundedBuf
- limit:Integer
- size:Integer

For each attribute, a label identifying the attribute is specified, optionally followed by a class name to define the type of the attribute.

---

[4] Note that an attribute is a fully-fledged object, in contrast with the notion of attributes as pure data values that is adopted by some analysis and design methods (e.g. [Coad 91a,b] and [Rumbaugh 91]).

Figure 4.2.4 above shows the objects in the Company subsystem, extended with parts and attributes. The figure shows three objects, respectively of class Machine, BoundedBuf and SalesDept (by convention, class names are capitalised). The Machine object contains an attribute volume and a nested instance of class Fuel. The storage object has two attributes, respectively limit and size, both instances of class Integer.

Defining Message Interfaces

The final aspect specified by basic object diagrams is the external interface of objects. The interface of an object is determined by the messages that it -potentially- interchanges with other objects. This consists of both the messages that the object accepts (input-interface), and the messages that an object sends to other objects (output-interface). The messages on the interface of an object are defined in the basic object diagram by attaching a bar with the name of the message to the side of the rectangle (this bar is also referred to as a -message-*stub*). The outgoing messages are written with a dash above it, for example '$\overline{\text{message}}$'. By convention, input messages are usually drawn on the left-hand side of the object rectangle, and outgoing messages usually on the right-hand side.

The notation is illustrated by the following figure. In this diagram, the sales object accepts the two messages request and sell, and sends the messages get and invoice to other objects.

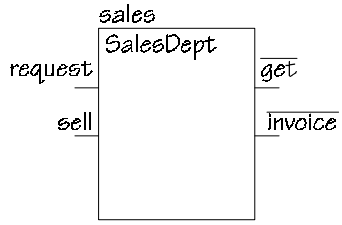


Figure 4.2.5 Definition of the interfaces for both incoming and outgoing messages.

It is now possible to describe the interface of objects, irrespective of their context. This interface is primarily determined by the responsibilities each object has. Or, stated differently, by the services the object is expected to deliver. The output interface is mainly determined by the activities the object must perform to realise its responsibilities.

Abstract Syntax of BODs

For each of the diagram types that appears in this chapter, we will give a formal definition of its components. This serves several purposes; as a precise specification of the components of the notation, but can also be used later for two purposes: the first is to show that the various diagrams can be combined in a consistent manner, and that every change in one type of diagram has a well-defined effect on other types of diagram. The second purpose is to define the input for an algorithm that generates composition-filters specifications from diagrams.

The abstract syntax of the basic object diagrams is as follows, using the METANOT notation defined in the appendix. The entities that are specified here are tagged with the type of the diagram, in this case 'BOD'.

$BasicObjectDiagram \stackrel{def}{=} LabeledEntities_{BOD}{}^{*}.$

$LabeledEntities_{BOD} \stackrel{def}{=} label{:}Identifier;\ entity{:}Entity_{BOD}.$

$Entity_{BOD} \stackrel{def}{=} ObjectCollection_{BOD} \mid Object_{BOD}.$

$ObjectCollection_{BOD} \stackrel{def}{=} objectDef:Object_{BOD}.$
$Object_{BOD} \stackrel{def}{=} className:Identifier; attribs:Attributes_{BOD}; interface:Interface_{BOD};$
$parts:Parts_{BOD}.$
$Attributes_{BOD} \stackrel{def}{=} Attribute_{BOD}*.$
$Attribute_{BOD} \stackrel{def}{=} label:Identifier; type:Identifier.$
$Interface_{BOD} \stackrel{def}{=} input:MessSet_{BOD}; output:MessSet_{BOD}.$
$MessSet_{BOD} \stackrel{def}{=} MessSpec_{BOD}*.$
$MessSpec_{BOD} \stackrel{def}{=} selector:Identifier.$
$Parts_{BOD} \stackrel{def}{=} LabeledEntities_{BOD}*.$

4.2.3 Object Structure Diagrams (OSDs)

Object Structure Diagrams are used in the initial phase of the design. They define the structure of the system and the static relations between objects. In this subsection we will define the notation for object structure diagrams and its semantics.

In the object-oriented paradigm, the relationships between objects can be classified into three categories: reuse, aggregation or message connection. Of the three categories, the reuse and part-of relation are static[5] relationships, whereas message connections deal with the typical dynamic issue of interaction through message sending.

Object structure diagrams (OSDs) deal with the static structure of an application only: the reuse and aggregation relations between objects. As we distinguish two types of reuse relation, inheritance and delegation, the following three notations are available in OSDs:

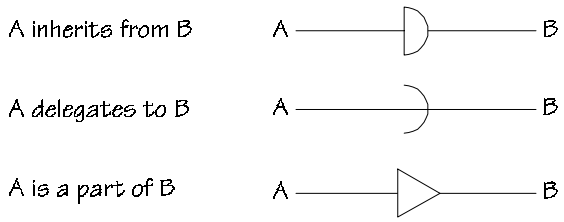


Figure 4.2.6 The three notations for structural relations.

An inheritance relation defines a specialisation-generalisation relation between two objects. Inheritance works at the class-level. The *inheriting* object (subclass) is a specialisation of the *inherited* object (superclass): it obtains all the properties of the superclass, but may add or refine some properties. A class may have any number of subclasses and may also have multiple superclasses (multiple inheritance).

Delegation is similar to inheritance in that it reuses predefined objects. The important difference, however, is that delegation reuses instances rather than classes. An instance has, apart from a message interface that can be reused (code sharing), an encapsulated state. Thus the use of delegation combines code sharing with data sharing.

The notation for an aggregation relation as shown in figure 4.2.6 defines objects as parts of other objects. This is semantically exactly the same as the nesting of objects and the

5 This is depending on the precise semantics of the object model: in the conventional object-oriented model, both part-of and inheritance relations are completely static. The composition-filters model supports dynamic inheritance and delegation. These dynamic aspects is not dealt with in OSDs.

definition of attributes in BODs. This redundant notation for nested objects is provided purely for pragmatic reasons, of which the main motivation is to support convenient hand-drawing of diagrams, in particular when iterating.

In the following figure an example of an OSD is shown:

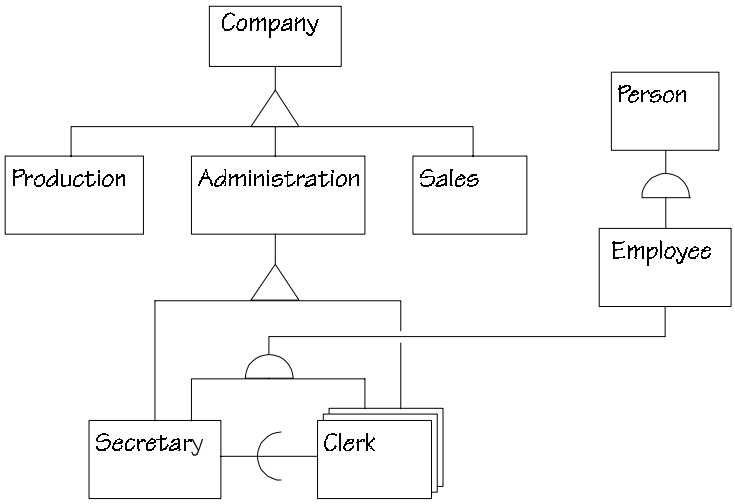


Figure 4.2.7 An example of an OSD.

In this example a high-level description of a company is contrived. Each of the relation types appears in the model. As is often the case, the aggregation relations form a suitable skeleton around which the application is modelled[6]. The Company object consists of three departments, respectively Production, Administration and Sales. The Administration consists of a Secretary and a set of Clerk objects. Note that in this example, no labels have been added to the objects at all.

Both the Secretary and the Clerks inherit from the class Employee, which in turn inherits from class Person. The Clerk objects all delegate to the Secretary. Delegation is useful in this case if the secretary object performs actions for the clerks that require a shared state, such as making appointments and arranging meetings.

OSDs versus Basic Object Diagrams

The three relations that we discussed in this subsection are conventional object relations. The most important reason for introducing these in OSDs is that they are well-known and intuitive. As was mentioned earlier, the main motivation for OSDs is to construct an initial model of the application However, the composition-filters computation model adopts a different point of view on these relations, focusing on composition. A single reuse relation replaces the inheritance and delegation relations in OSDs. It will now be shown how these types of relations can be mapped onto each other.

Until now, the only type of structural relation that we considered in the BOD is the part-of relation (through nesting of objects). The second type of structural relation is a reuse relation; this can applied to realise either inheritance (single or multiple), or pure delegation.

---

6 This is especially true as aggregation relations can often model easily identifiable real-world relations, such as physical or organisational containment.

Such reuse relations can be seen as a means of redirecting (delegating) received service requests (messages) to either a superclass or a delegated object. This is specified in the object diagram notation by drawing a line from the interface of the reusing object to the interface of the reused object as is illustrated in the following figure:

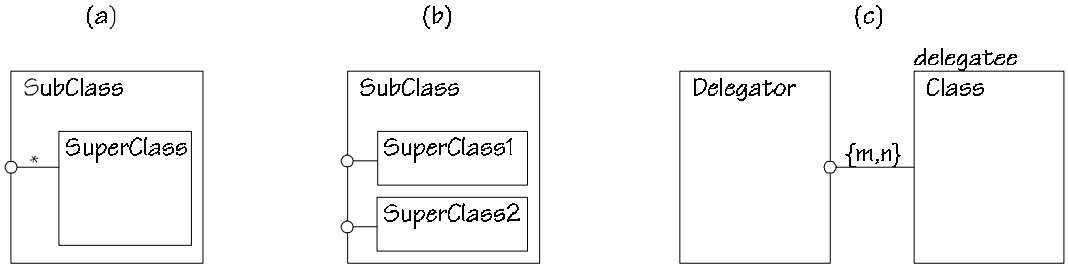


Figure 4.2.8 Three examples of reuse relations: (a) single inheritance, (b) multiple inheritance, (c) delegation.

A reuse relation is expressed by connecting the two participating objects with a line. On one end of the line, a bullet is connected to the *reusing* object[7]. The other end of the line represents the *reused* object. The line can be labelled with an expression indicating the set of messages that are to be reused. The '*' wild card is available to designate all messages that are provided by the signature of the reused object. If the reuse relation is not labelled, the wild card is assumed.

The figure above shows three examples which demonstrate the application of the reuse relation. In example (a), conventional inheritance is shown: the SubClass object reuses all messages provided by the SuperClass object. As each SubClass object encapsulates a SuperClass object, it automatically contains a private copy of the instance variables that are defined by SuperClass. Thus, the SubClass objects reuse both the methods and the instance variables[8] of the SuperClass object, which is exactly the definition of inheritance.

The second example, (b), demonstrates multiple inheritance. In this case, the SubClass object has two reuse relations: one with object SuperClass1, and one with object SuperClass2. As no message set is specified, wild cards are assumed. Thus, the object fully inherits from both SuperClass1 and SuperClass2. Note that possible conflicts, which may occur if both superclasses support messages with equal names, cannot be resolved in this graphical notation[9].

The last example, (c), demonstrates delegation. The Delegator object reuses the messages m and n from the delegatee object (the label is added to stress the fact that the delegatee object is a unique instance). Thus the Delegator object -partially- shares the behaviour

---

7 This emphasises that the reuse relation is specified by the reusing object.

8 As well as all other components of an object in the composition-filters computation model, which we ignore here for brevity.

9 A solution to this would be to define an ordering on the reuse relations, e.g. counter-clockwise from the left upper corner. However, this is quite tricky and cannot solve all conflicts satisfactory (as one may not want all messages of one superclass to prevail over all the other).

(methods) and the internal state of the delegatee. Note that multiple instances of the Delegator object will all delegate to the same instance labelled delegatee.

The reuse relation presented here is actually a delegation relation. It was previously discussed in the context of Dispatch filters in section 2.4.3 how delegation can be used to simulate inheritance and pure delegation. In this discussion of composition filters, the notation 'target.selector' is used in the specification of a Dispatch filter. In the object diagram notation, a semantically equivalent line is drawn from the interface of an object to the target object, and the selector is written next to this line.

The following figure shows the notations in OSDs together with the equivalent versions expressed in the basic object diagram notation. For each relation type, the OSD notation is shown on the left, and the composition-filters equivalent is shown on the right hand side of the arrow:

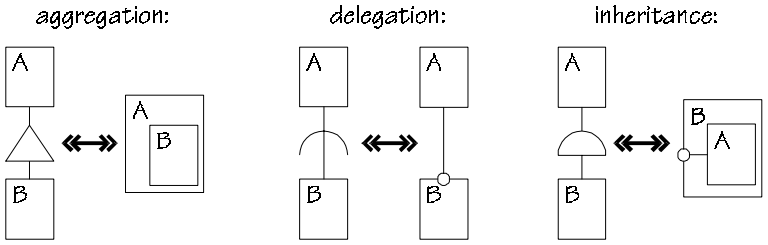


Figure 4.2.9 The mapping between the OSD notation and the BOD notation.

Note that both notations can be freely interchanged. As a result every change in one type of diagram has well-defined consequences on diagrams of another type (assuming they represent the same application). In full object diagrams (FODs), both notations can be mixed freely. It is suggested, however, that the developer gradually moves to a fully composition-based model of the application when objects are considered with more detail.

Abstract Syntax of OSDs

The abstract syntax of object structure diagrams is defined here. Because the formal model for describing abstract syntax has no notion of inheritance, we have to redefine everything that differs from the basic object diagrams. However, for parts that are identical in BOD and OSD, we can simply refer to the definitions in the abstract syntax of the BOD. This is the case for the *LabeledEntities* and *Object*. The subscripts indicate where the definition can be found. New parts in rules that are redefined are marked as bold.

First the object structure diagram is defined as a combination of a set of objects and a set of relations between objects:

$ObjectStructureDiagram_{OSD} \stackrel{def}{=} objectSet{:}LabeledEntities_{BOD}; relSet{:}Relations_{OSD}$.

The *Object*s in the *objectSet* are a slight extension of the previously defined $Object_{BOD}$, therefore *Object* is redefined here. Note that the reuse relation is considered to be a part of definition of the objects, whereas the structural relations are modelled independent from objects.

$Object_{OSD} \stackrel{def}{=} className{:}Identifier; attribs{:}Attributes_{BOD}; interface{:}Interface_{BOD}$;
$parts{:}Parts_{BOD}$; $\mathbf{reuse{:}ReuseRels_{OSD}}$.

$ReuseRels_{\mathbf{OSD}} \stackrel{def}{=} ReuseRel_{\mathbf{OSD}}{}^*$.

$ReuseRel_{\mathbf{OSD}} \stackrel{def}{=} mess{:}MessSet_{\mathbf{OSD}}$; $obj{:}ObjectRef_{OSD}$.

Then the set of structural relations is defined, split into the three types: inheritance, delegation and aggregation respectively:

$Relations_{OSD} \stackrel{def}{=} Relation_{OSD}^*$.
$Relation_{OSD} \stackrel{def}{=} Inheritance_{OSD} \mid Delegation_{OSD} \mid Aggregation_{OSD}$.
$Inheritance_{OSD} \stackrel{def}{=} superclass{:}ObjectRef_{OSD}; subclass{:}ObjectRef_{OSD}$.
$Delegation_{OSD} \stackrel{def}{=} delegatee{:}ObjectRef_{OSD}; delegator{:}ObjectRef_{OSD}$.
$Aggregation_{OSD} \stackrel{def}{=} part{:}ObjectRef_{OSD}; whole{:}ObjectRef_{OSD}$.
$ObjectRef_{OSD} \stackrel{def}{=}$ (*this is a reference to another object in the current object diagram*) .

The representation of object references in the abstract syntax is somewhat problematic: in a graphical notation this is conveniently done by drawing connecting lines to and from objects. In the abstract syntax we have no means to elegantly express such relations. Therefore we introduce the compound *ObjectRef*, which is a reference to instances of the *Object* compound. We do not show here how this reference is defined.

4.2.4 Object Interaction Diagrams (OIDs)

During the execution of an object-oriented program, objects cooperate and interact with each other to fulfil their responsibilities. A mere static description of individual objects does not capture this. Object interaction diagrams (OIDs) provide a means to describe certain configurations of objects, including the interaction connections between them. Multiple OIDs, where each diagram describes a separate configuration, can be used to cope with objects that dynamically change interaction patterns, for instance due to changing contexts.

Message connections are defined by (possibly two-headed) arrows between the participating objects. These arrows connect at the object, replacing the 'stubs' that define the message interface of an object. The direction of the arrow indicates the message flow. A line with no arrows attached to it, or a two-headed arrow, indicates that the same messages are exchanged in both directions.

The message connections are labelled with the message(s) that are exchanged between the objects. Obviously, there is no need to write the name of the message on both sides of the message connection. The following figure illustrates the notation for message connections:

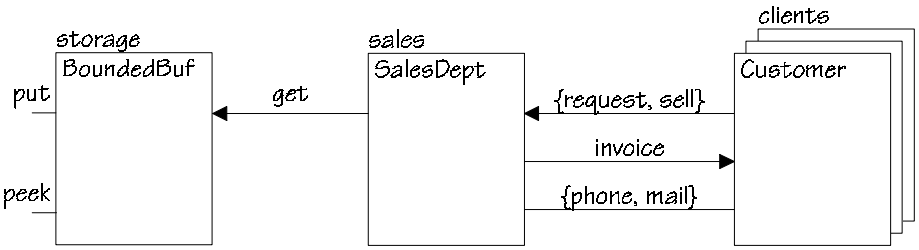


Figure 4.2.10 Message connections of the sales department in the company example.

In this example, which concentrates on the interactions of the sales object, all three types of message connections are applied. The object has two connections defining outgoing messages, respectively get to the storage object, and invoice to the clients. From -each of- the clients two possible messages may be received: request and sell. In addition, there is a bi-directional message connection between clients and sales. Both may send, as well as receive, phone and mail messages. From this diagram we can conclude that the SalesDept

class should accept any of the messages request, sell, phone, and mail. In addition, the class sends the messages get, invoice, phone, and mail to other objects.

There is a subtle notational convention for objects in collections that send messages to themselves. This is demonstrated by the following collection of Clerk objects:

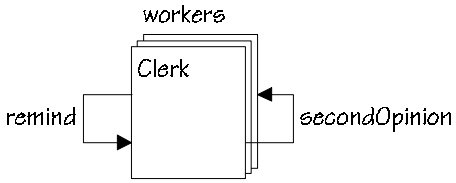


Figure 4.2.11 Sending messages to collection members.

The elements in this collection send remind messages to themselves. For a message to a collection element itself the arrow ends points to the left or bottom side of the collection symbol. The secondOpinion message is sent to *another* element in that collection. This kind of message connection is drawn such that the destination arrow is on the right or top side of the collection symbol. To emphasise the difference, the tail of the arrow starts on the inside of the collection symbol, whereas the head of the arrow points to the outside of the symbol. Note that the latter message connection is potentially bi-directional: each element in the collection can inherently both send and receive the message.

The object interaction diagrams, as explained in this subsection, provide the developer with a road map to the dynamic behaviour of objects. As OIDs reveal the locations where dynamic interactions between objects take place, they are an important instrument in the development of dynamic systems.

Abstract Syntax of OIDs

The abstract syntax of the object interaction diagrams is as follows:

$$ObjectInteractionDiagram \stackrel{def}{=} objectSet{:}LabeledEntities_{BOD};\ interaction{:}MessConns_{OID}.$$
$$MessConns_{OID} \stackrel{def}{=} MessConn_{OID}{}^{*}.$$
$$MessConn_{OID} \stackrel{def}{=} direction{:}Direction_{OID}\ ;\ from{:}ObjectRef_{OSD}\ ;\ to{:}ObjectRef_{OSD}\ ;\ mess{:}MessSet_{BOD}\ .$$
$$Direction_{OID} \stackrel{def}{=} Right \mid Left \mid BiDir.$$

It is clear to see that the OID is a straightforward extension (syntactically, that is) of the BOD: it adds the message connections -the *interaction* component- to the set of objects as defined by the BOD.

4.3 State Composition Diagrams

In this section we will explain the basics of *State Composition Diagrams* (abbreviated as 'SCDs'). SCDs describe the dynamic aspects of objects, focusing on the synchronisation of messages on the object interface in relation with the internal state of the object. Object diagrams form the skeleton in which the state composition diagrams are embedded. An object diagram describes the static structure of objects: nested objects, the method interface, reuse relations and message connections between objects.

After identifying which objects the system consists of, what their respective structure is, and how they are interrelated, we will now focus on the dynamic aspects of objects. The basic approach is to look at the internal state of an object, provide abstractions of that state, and associate synchronisation of messages with states. We adopt a visual formalism that resembles state-transition diagrams for modelling these dynamic aspects. The graphical notation was designed with both hand-drawing and computer-supported tools in mind.

4.3.1 A Brief Introduction to State Composition Diagrams.

A state composition diagram essentially describes the synchronisation of messages on the interface of an object. An object diagram provides the context for the state composition diagram. Although SCDs can be constructed independently from object diagrams, we consider it important that the developer is aware of the context of the object for which synchronisation is described.

As an example, consider the following object diagram of an application:

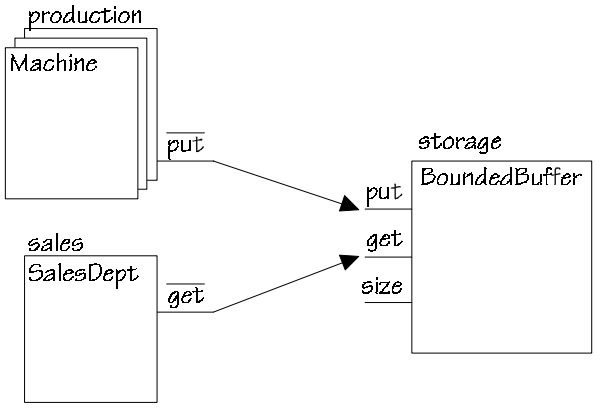


Figure 4.3.1 The object diagram of a producer-consumer system.

This object diagram shows a collection of Machine objects (producers), sending put messages to the storage object (a buffer), and a SalesDept object sending get messages. The bounded buffer is a typical server-type of object: it only receives, and serves, requests from other objects, without sending messages to external objects.

From this object diagram we can obtain important information regarding the behaviour we require from the bounded buffer object. First of all, as there are a (possibly large) number of clients, interleaving put and get messages are to be expected. In general, even without multiple client objects, interleaving messages are possible, as objects may be internally

concurrent. We will for now assume that the bounded buffer object allows no internal concurrency (this is the 'default' assumption in the method). Allowing internal concurrency within the buffer object can be done later as a refinement step. For now we are primarily concerned with the problems of invocations of get messages on an empty buffer and put messages on a full buffer: synchronisation of messages.

Obviously, the synchronisation of these put and get messages depends on the internal state of the buffer object (in particular, of the number of elements that are currently stored in the buffer and the number of free places). Therefore, we take a closer look at the bounded buffer object:

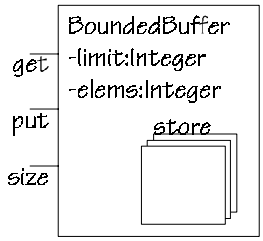


Figure 4.3.2 The internal structure of class BoundedBuffer

The bounded buffer object has two attributes: limit and elems, both integers. The limit attribute denotes the maximum amount of elements the buffer can store, the elems attribute indicates the actual number of elements in the buffer. The buffer contains a single part, labelled store. This represents the storage structure in which the elements are accumulated. We do not bother with the actual structure of this storage here.

> Note that we can express synchronisation either in an abstract way, through the method interface (the size method in this case), or by looking at the internal structure of the object (the elems attribute). The first approach is preferable from the software engineering point of view, as it abstracts from the implementation of the buffer. However, it is less intuitive and less pragmatic, as it requires a preliminary knowledge as to which information regarding the state of the object is relevant on the interface[1] of the object. In addition, an interface is flat and static, and does not properly reflect the dynamics of the object (such as the interaction between nested objects and nested collections of varying size). Therefore, in our examples we will often investigate the internal structure of an object before defining the synchronisation constraints of that object. Note however, that the methodology does address the issue of expressing synchronisation constraints in a more abstract way: this is addressed in a separate, iterative step.

We will now show how to construct a graphical representation of the synchronisation constraints of the object. The approach is to draw a diagram that resembles a state-transition diagram. This diagram expresses the relations between the state of an object (which is abstracted as a set of state identifiers) and a set of events (i.e. the messages that are allowed to be dispatched) that are specified by the diagram. We demonstrate this with the bounded buffer example[2] in figure 4.3.3.

---

1 Here, we do not make the distinction between an external and an internal interface.

2 We do not show the simplest possible solution here, which requires only two states (nonEmpty and spaceLeft) for illustrative reasons: states are more than just method guards, as they are first

In this diagram, three states are drawn as rounded rectangles: Empty, Partial and Full. By convention, state names start with a capital. The state icon contains the identifier of the state and, between round brackets, a Boolean expression specifying that state. Suppose that with a state identifier s_i, a Boolean expression b_i is associated, as expressed by $\langle s_i , b_i \rangle$. Then the object is 'in' state s_i only if b_i is *true*: $b_i \supset s_i$. If we assume that an object o has defined a set of states S_O, then at a particular moment in time, the object is 'in' a set of states S_{cur}:

Definition 4.3.1

A state s is valid for object o with states S_o if, and only if, $s \in S_{cur}$

where $S_o \stackrel{def}{=} \{ \langle s_1 , b_1 \rangle, \langle s_2 , b_2 \rangle, \ldots , \langle s_n , b_n \rangle \}$;

$S_{cur} \stackrel{def}{=} \{ \langle s, b \rangle \in S_o \mid b \}$

Thus, an object can be in multiple states; however, we prefer to model this through a combination of SCDs (as will be explained later), rather than allowing multiple states within a single SCD to overlap. Note that this constraint exists for pragmatic reasons only, not for technical reasons.

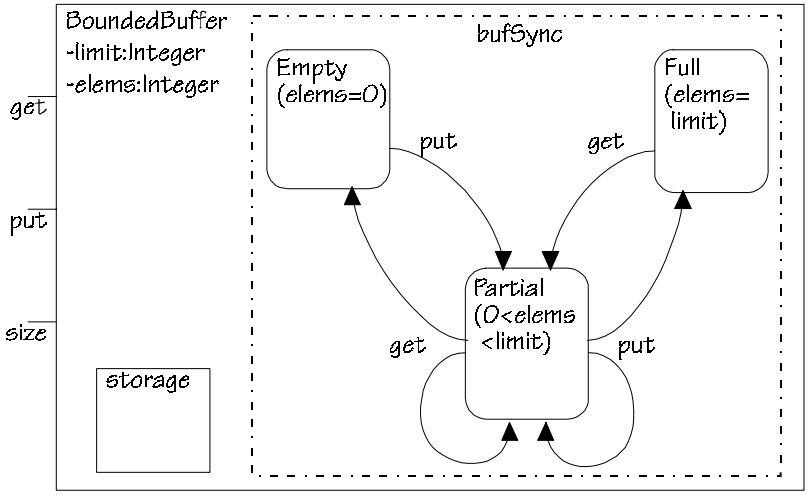


Figure 4.3.3 The SCD for BoundedBuffer.

Between the states, arrows are drawn which indicate the events that transform the object from one state to another. In this example, the possible events are put and get. Events are represented by messages[3]; when a message is dispatched, it may affect the state of the object, causing the transition from one state to another. A transition may have the same start and ending state.

The crucial information in the SCD is the set of outgoing transitions for each particular state, as *no other transitions are allowed*. This effectively describes the synchronisation constraints of the object: when a message is received and the object is in a state with no outgoing transition for that particular message, the message will be blocked. Only when the

---

of all abstractions of the state of the object. They can be associated with (sets of) messages to express synchronisation constraints on these messages.

3 We will use the terms message and event interchangeably.

object reaches a state with an outgoing transition for that message, the message is *acceptable*, i.e. it can be dispatched. It is important to note that the constraints only apply to the messages that are explicitly covered by the diagram:

Definition 4.3.2:

A message *m* is acceptable at a particular moment if, and only if, there is an outgoing transition $\langle s_{from}, m, s_{to} \rangle$ for *m* from a valid state, i.e. if $s_{from} \in S_{cur}$, or $m \notin M$, where *M* is the set of messages covered by the diagram; $M \stackrel{def}{=} \{n \mid \forall \langle s_{from}, n, s_{to} \rangle \in \text{SCD}\}$.
(This definition will be refined later in this section.)

Returning to our example; it expresses the synchronisation constraints for the put and get messages: as we can see in the SCD in figure 4.3.3, the put message has outgoing transitions from the Empty and the Partial states, and the get message has outgoing transitions from the Partial and the Full states. This means that the synchronisation of the BoundedBuffer is such, that no put message will be dispatched for a full buffer, and no get message for an empty buffer, as such transitions are not in the SCD.

It may seem that SCDs are not the most concise way of specifying synchronisation constraints. Indeed, when the required synchronisation constraints are known, SCDs may not be the most attractive notation. However, the concept of SCDs is introduced as a methodological tool to *find* synchronisation constraints, rather than *define*. Thus, aspects such as intuitiveness and pragmatics are important design considerations. We believe that the SCD-model is suitable for our purposes as it addresses intuitiveness (state-transition model), and presents the relevant contextual information for distilling synchronisation constraints. The presentation of the activities (events/messages) and the state of the object is likely to be a more fruitful base for thinking and reasoning about synchronisation than a declarative presentation of pure constraints would be. The extraction of the actual synchronisation constraints from the SCD is a side-effect rather than a primary concern during SCD construction.

4.3.2 Transitions in State Composition Diagrams

In our previous example, we only showed the basic type of transition, labelled with the name of a single message. However, a number of variations to this exist. Firstly, just as in object diagrams, there is the distinction between incoming and outgoing messages: a dash above the message selector, e.g. "$\overline{\text{put}}$" indicates that it concerns an outgoing (sent) message, rather than an incoming (received) message. Secondly, instead of a single message selector, a set of message selectors can be defined. For example, "{put, get}" is equivalent to two separate transition arrows, labelled with put respectively get.

Wild cards & Exclusion

Instead of fixed message selectors, wild cards can be used. This is very important, as it specifies open-endedness[4]. There are two forms for wild cards: a plain wild card "*" which simply means that any message can make the transition, and an *exclusion*, "~*m*", which means that any message, except *m* can make the transition. Likewise, "~$\{m_0, m_1, \ldots, m_n\}$" means that any message except those in the set $\{m_0, m_1, \ldots, m_n\}$ can make that transition.

4 We will discuss the use of wild cards more extensively in the context of design considerations.

Transition Conditions

Apart from the selectors of the message(s) that a transition represents, a transition arrow may be labelled with a condition. This condition must be satisfied for the transition to be allowed. Conditions are denoted between square brackets preceding the message selector (set). For example, a transition arrow with the label "[gas>0]drive" indicates that when the expression "gas>0" evaluates to *true*, the transition *drive* is allowed, but only when the originating state is valid! Expressed in a more formal way:

Definition 4.3.3: (refinement of definition 4.3.2)

Assume the following definitions:

$t \stackrel{def}{=} \langle s_{from}, cond, e, s_{to} \rangle$, t is a transition, where

s_{from} and s_{to} are respectively the originating and the destination state,

cond is the transition-condition,

e denotes the expression that describes the messages that are allowed by this transition,

$M \stackrel{def}{=} \{n \mid \forall \langle s_{from}, cond, e, s_{to} \rangle \in SCD, n \text{ in}^5 e\}$: the set of messages covered by the SCD.

Then a message *m* is acceptable at a particular moment either when transition *t* can occur or when the message is not covered by the diagram, i.e. $m \notin M$.

The transition *t* will occur for a message *m* if, and only if the following three conditions are satisfied:

(1) $s_{from} \in S_{cur}$, (the object is in state s_{from},)

(2) *cond*, (the associated condition is valid)

(3) $isWild\ card(e) \vee (\neg isExclusion(e) \wedge m \in e) \vee (isExclusion(e) \wedge m \notin e)$
(the transition allows for the message selector of *m*)

It is important to note that the additional of conditions to transitions is purely available for convenience of describing the dynamic behaviour. The same behaviour can be achieved with only states and transitions without conditions.

Mutual Exclusive Transitions

As a notational convenience, the notion of mutual exclusive transitions is introduced. Within a single object, only one mutual exclusive transition can be performed at a time. This prohibits potential conflicts between concurrently executed transitions. The mutual exclusive transitions are drawn in the SCD as thick, or as double arrows:

Figure 4.3.4 Two alternative notations for mutual exclusive transitions.

Obviously, this is useful for objects with intra-object concurrency only (the notation for objects that allow intra-object concurrency will be explained soon), where it allows for convenient protection of resources for transitions with side-effects. The transition is executed as a critical section.

5 "*n* in *e*" means that *n* is either syntactically specified by the expression *e*, or *e* contains a wild card.

4.3.3 Composition of SCDs

The construction of SCDs that capture the full dynamics of complex objects may result in a combinatorial explosion of states [Harel 87]. Modularisation of state compositions supports the management of complex systems, and allows for improved reuse and extensibility (see section 3.2). SCDs offer two levels of state composition modularity: object composition and SCD composition.

Object composition is expressed by the object diagrams and encapsulates the details of the SCDs. SCD composition occurs within a single object, and composes the (synchronisation-) behaviour of the object from several SCDs:

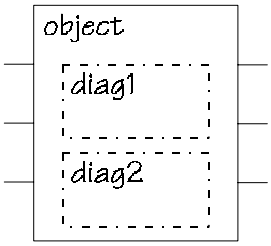


Figure 4.3.5 SCD composition.

An SCD is bounded by a dashed rectangle. Each diagram is labelled for future reference (respectively diag1 and diag2 in the figure). The label is placed just inside the dashed rectangle, at the top.

Composition of SCDs is *restrictive*: this means that the two (or more) diagrams restrict each other. Thus, for a message to be accepted, it must be accepted by all SCDs. This can be thought of as an AND operation on SCDs. However, from the specification of SCDs (definition 4.3.3) it follows that an SCD defines only constraints for a restricted set of messages. Thus, two SCDs that deal with different messages do not restrict each others synchronisation constraints. Such a case could be judged as an OR-ed composition of SCDs.

The following figure shows how SCDs can be reused from other objects:

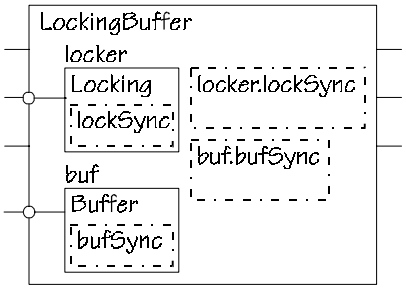


Figure 4.3.6 SCD reuse demonstrated through the LockingBuffer.

In this example the class LockingBuffer defines multiple inheritance from classes Locking and Buffer, respectively. The synchronisation of the LockingBuffer is defined as a composition of the two SCDs that are defined by these classes. This is specified by referring to the labels of the internal objects (locker and buf) that represent the superclasses and the labels of the respective SCDs (lockSync and bufSync). The reuse of SCDs differs from the

reuse of methods, instance variables, etcetera, as the reused SCDs are interpreted in the context of the reusing object[6].

4.3.4 Internally Concurrent Objects

As stated earlier, we use the object diagram notation for embedding the SCDs. There is one notational extension to object diagrams, however. This is for the purpose of defining intra-object concurrency. In the methodology, we follow the same approach with respect to intra-object concurrency that we follow in the language: by default, every object is mutual exclusive. Through an additional notation, we can remove this constraint on a per-object basis. Thus, we introduce an additional notation for objects that allow for intra-object concurrency:

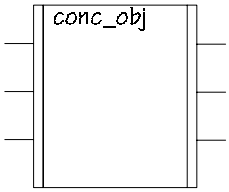


Figure 4.3.7 The notation for objects that allow intra-object concurrency.

The additional bars symbolise parallelism, and are suitable to be added to an existing diagram. The latter is important as it allows intra-object concurrency to be added during iteration, rather than at the first moment of drawing the object diagram.

4.3.5 Annotations

We now introduce two more notational elements; *virtual transitions* and *action annotations*. These elements differ from the previously discussed elements by being annotations to the SCD. This means that rather than having a concrete realisation associated with them, they are considered to be only hints, or helping aids for the designer.

The first type of annotations are the *virtual transitions*. Virtual transitions are transitions (i.e. state changes) that take place without the explicit occurrence of an event (i.e. without specifying a message that causes the transition). Such a transition may still be drawn in the SCD for the clarity of the design, although it is not necessary. Virtual transitions are drawn as a dashed arrow:

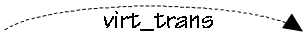


Figure 4.3.8 The notation for virtual transitions.

The second type of annotations are the *action annotations*. Action annotations associate actions, or causal relationships with transitions and states. They correspond to the actions as found in the Mealy, respectively the Moore model for state machines. The notation for action annotations is illustrated by the example in figure 4.3.9.

In this example of an engine, two states are shown, that are abstract representations of the amount of fuel: when the engine is in state *Empty*, there is no fuel available in the system

6 Note, however, that the implementations of the conditions cq. states in an SCD depend on the object where they were defined.

(defined by the condition "fuel=0"), otherwise the system is in state *Filled* (when "fuel>0" is valid). Two events are possible in the system: *go*, which is only possible when the system is in state *Filled*, and which has an associated action "fuel--", indicating that the amount of fuel decreases. Every time the *go* event occurs, the amount of fuel is diminished by 1.

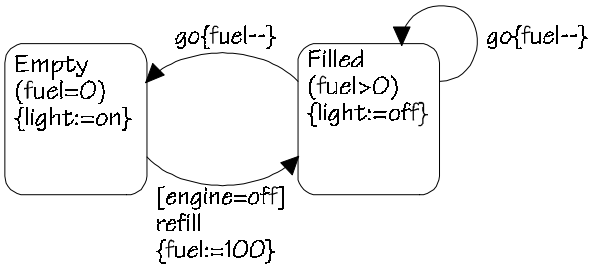


Figure 4.3.9 An example showing the use of action annotations.

The other event is *refill*, which may only occur when the engine is in state *Empty*, and the 'engine is off' (as indicated by the expression between the square brackets). In this case the action "fuel:=100" is performed. Upon entry of a state, an associated action is performed as well: when entering the state *Empty*, a warning light is turned on ("light:=on"), and when entering the *Filled* state, the light is turned off ("light:=off").

It is important to note that at this point, the action annotations are not supported by the methodology, other than through a few hints. We do not want to use the annotations for the generation of code. As was argued in section 4.1, we feel that this might interfere with the behavioural specification steps of the methodology: On one hand, SCDs should not be used for the specification of complete method bodies, and on the other hand, the action annotations in SCDs cannot be incorporated in the proper method bodies of the object in a straightforward and controllable manner.

Parallelising Transitions

As the notion of synchronisation is closely related to concurrency or parallelism, it is advantageous to express some forms of creating concurrency in SCDs. We have already seen this in the form of parallel objects. The creation of additional concurrency in the system is possible through the application of the early return construct. In the context of SCDs this means that a part of the execution of the transition is done after the transition is 'finished'. This is especially important for the assumptions that are made when the object is in the target state.

As an example, consider a transaction *T* from state *A* to state *B*. When this transition is 'finished', i.e. it returns a reply, there are two possible situations. The first is that the object is already in state *B*, the second is that the object has not reached that state. The following notation is available to express either of these cases:

Figure 4.3.10 Parallelising transitions: early fork (l) and late fork (r).

In an *early fork* transition the object will not necessarily be in the target state when the transition returns a reply. In the *late fork* transition on the other hand, it is ensured that first the state change of the object is effectuated, before the transition returns a reply. This

mechanism should be applied with care, to avoid inconsistencies due to concurrent activities within the same object.

4.3.6 An Abstract Syntax for State Composition Diagrams

State composition diagrams are completely independent from object diagrams, except for the designation of objects as 'concurrent'. The abstract syntax specification of the state composition diagrams only is given below:

$SCDiagram_{SCD} \stackrel{def}{=} SCDunit_{SCD}^*.$
$SCDunit_{SCD} \stackrel{def}{=} label{:}Identifier;\ states{:}States_{SCD};\ transitions{:}Transitions_{SCD}.$
$States_{SCD} \stackrel{def}{=} State_{SCD}^* .$
$State_{SCD} \stackrel{def}{=} id{:}Identifier;\ spec{:}Expression_{CFM} .$
$Transitions_{SCD} \stackrel{def}{=} Transition_{SCD}^* .$
$Transition_{SCD} \stackrel{def}{=} PlainTrans_{SCD} \mid MutexTrans_{SCD} \mid VirtTrans_{SCD} .$
$PlainTrans_{SCD} \stackrel{def}{=} spec{:}TransSpec_{SCD};\ cond{:}Expression_{CFM} ;\ action{:}Expression_{CFM} .$
$MutexTrans_{SCD} \stackrel{def}{=} spec{:}TransSpec_{SCD};\ cond{:}Expression_{CFM} ;\ action{:}Expression_{CFM} .$
$VirtTrans_{SCD} \stackrel{def}{=} spec{:}TransSpec_{SCD};\ cond{:}Expression_{CFM} .$
$TransSpec_{SCD} \stackrel{def}{=} excl{:}ExclSpec_{SCD};\ set{:}MessSet_{SCD}.$
$ExclSpec_{SCD} \stackrel{def}{=} Exclude_{SCD} \mid Permit_{SCD} .$
$MessSet_{SCD} \stackrel{def}{=} IDSet_{SCD} \mid Wild\ card_{SCD} .$
$IDSet_{SCD} \stackrel{def}{=} Identifier^*.$

At various locations a reference to $Expression_{CFM}$ can be found: this stands for a message expression according to the composition filters model. This abstract syntax refers to the specification of the composition filters model (CFM), as was given in chapter 2.

After introducing the notations for specifying objects in the previous section, and for defining the synchronisation constraints of objects in this section, the next section will introduce a software development method. The method provides a number of method steps, guidelines and hints to support the actual derivation, invention and specification of objects and their synchronisation constraints.

4.4 The Method

4.4.1 Introduction

In this section a method for software development is proposed. The method can be roughly divided into two parts: in the first part the conventional methodology for object-oriented software development is adopted for modelling the static aspects of objects. Although notation and emphasis are different, the approach is similar to conventional object-oriented methods, as presented in e.g. [Rumbaugh 91], [Wirfs-Brock 89, 90a], [Coad 91a, 91b], [Booch 90] and [Jacobson 92]. Therefore, we pay less attention to this part of the method.

The second part of the method focuses on modelling the dynamic aspects of objects, with the main goal of coming up with the proper synchronisation specifications. Our aim is to let the software engineer define concurrent object specifications with the highest possible degree of reusability and extensibility. There is little or no related work with the same objectives.

Another important distinction between the method presented here and the bulk of the other methods is its suitability for automated support and the use of executable specifications. This is mainly due to the notations that were presented in the previous section. The translation is discussed in detail in the next section.

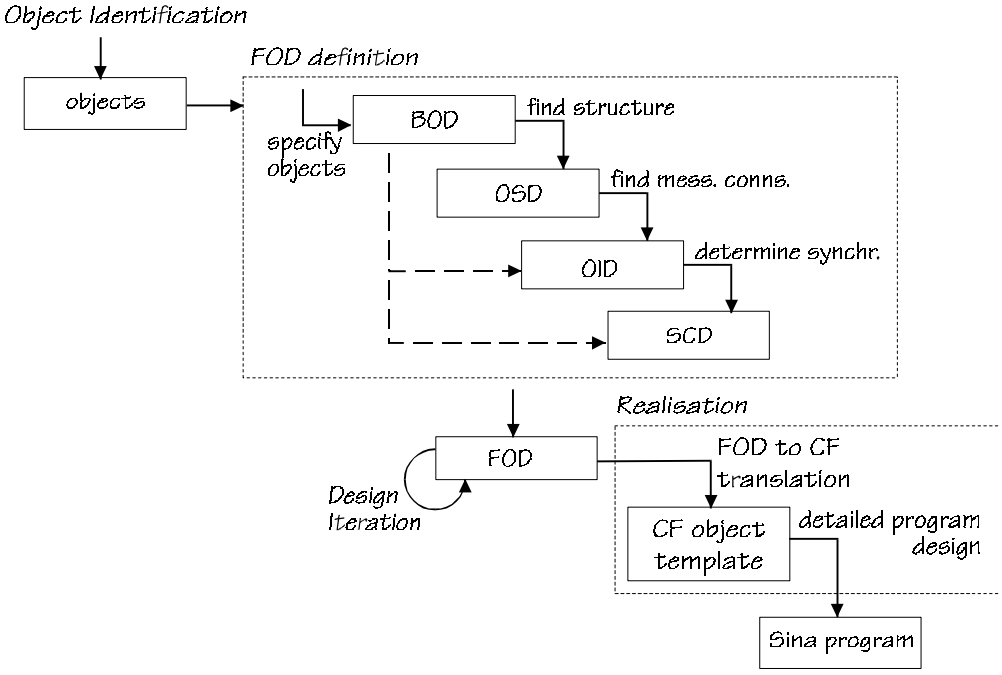


Figure 4.4.1 The dependence between components and steps of the method.

An Overview of the Method Steps and Components

Figure 4.4.1 shows the steps that the method is constructed of. Each step works on, or delivers, a component, for example a particular diagram. The solid rectangles represent these components, whereas the lines represent method steps.

Each step requires certain components to perform successfully and transforms these or creates new components. The figure above defines a precedence graph, indicating which step(s) must precede other steps, such that all required components for a step are available. The solid lines indicate the order in which the steps will be discussed in this section. This is also the advised order for the first iteration.

Mind that, although the figure may suggest a waterfall-like process, this is clearly not intended: the arrows indicate precedence *requirements* rather than a flow of control. Instead, we promote an iterative development cycle.

In addition to iterating over the various method steps by re-doing them, we devote a separate method step to the so-called *design iteration*. In this step a number of design considerations are made explicit. It is shown as a transformation process on the FOD.

We distinguish the following 4 phases in the overall development process. These phases by no means represent a division into parts of equal size, but separate activities that are rather different by nature. In particular phase B, the construction of an FOD, is an extensive phase which consists of multiple steps:

A. *Object Identification*: In this -initial- phase of the development, the objects that make up the subsystem are defined. This is accomplished by the method step with the same name.

B. *FOD Construction*: The identified objects are detailed, and structural and interaction relations among them are defined. Finally the synchronisation constraints are identified with the help of SCDs. The method steps of this phase are:
   - ❏ *Specifying Objects*: objects are specified with more detail, such as the message interface, attributes and nested objects.
   - ❏ *Define Structural Relations*: identify relations between objects such as inheritance, delegation and aggregation.
   - ❏ *Define Object Interactions*: define the -static- message connections between objects.
   - ❏ *Determine Synchronisation*: construct state composition diagrams.

C. *Design Iteration*: Consider the system with respect to a number of design rules, achieving increased extensibility and reusability is one of the main goals of this phase. The results are typically transformations and restructuring of the subsystem under consideration.

D. *Realisation*: When the design of the system is completed (or at least performed with sufficient detail), it must be realised into a working program. Two steps are distinguished in this phase:
   - ❏ *FOD to CF translation*: This is the algorithmic[1] translation of the object diagrams into composition-filters specifications. The result could for instance be a template for a Sina program.
   - ❏ *Implementation*: The gaps that remain in the generated template are to be coded in this step. The main examples of such gaps are the implementation of method bodies and the addition of some specific filters.

---

1 Preferably, but not necessarily automated.

In the remainder of this chapter we will describe each of the method steps with more detail.

The Description of Method Steps

In order to bring some structure to the description of the method steps, for each step a template describing the main characteristics of the steps is given. This template briefly gives the most important information about the step:

Step #: the number of the step with a descriptive name.
Input: the components that are required to perform the step.
Output: the new or modified components after the step.
Side-effects: the most common side-effects.
Description: a brief description of the step.
Hints: a list of (brief) hints and directions.
Substeps: a list with the substeps, these are worked out (including a template description) in subsequent subsections.

After this template, the description of the step and the hints for this step are worked out. the substeps are either worked out in a separate subsection, or briefly explained. In the first case the substeps are numbered, in the latter case they are presented as a bulleted list.

In parallel with the discussion of the method steps, the modelling of a well-known synchronisation problem will be demonstrated: the dining philosophers problem. The problem description is given in the next subsection. Note that the discussions of presented example are not intended to reflect an actual development cycle in chronological order; they should be considered as highlights from such a development cycle. After the description of each of the steps in the forthcoming subsections, the software development process is discussed.

4.4.2 The Running Example: Dining Philosophers

We will illustrate the steps of the method through a step-by-step construction of a solution to the dining philosophers problem. In this synchronisation problem *n* philosophers are sharing a bowl of rice and *n* chopsticks, as illustrated in figure 4.4.2. The philosophers alternately think and eat. Obviously, a philosopher needs two chopsticks for eating rice; the chopstick on his left and right hand side, respectively. The problem is to define synchronisation constraints for each philosopher such that no starvation will occur, assuming that each philosopher eats for a finite time.

Each philosopher goes through an (assumed eternal) cycle of thinking, then getting hungry, eating, and thinking again. Hungry philosophers can only start eating once they have acquired both chopsticks. The main danger threatening the philosophers is that a cycle will occur where all philosophers are hungry, hold one chopstick in, for example, their right hand, and are waiting for their left neighbour to hand the other chopstick. In such a situation, no philosopher can ever start to eat.

Several solutions for avoiding this deadlock problem are available; one solution is to pick only two chopsticks at a time, and if this is not possible, get neither of them (i.e. make picking the chopsticks an atomic transaction). Another solution (that we will adopt), proposed by Chandy and Misra [Chandy 84] is to ensure that no cycle as described above

can ever occur. They propose a so-called *hygienic* solution: Every chopstick can be either clean or dirty. A chopstick gets dirty when used for eating, and will remain dirty until it is cleaned. A philosopher does not give away chopsticks while eating. When not eating, a philosopher gives away dirty chopsticks to its neighbours, and cleans them just before handing over. A request for a chopstick that is clean is deferred.

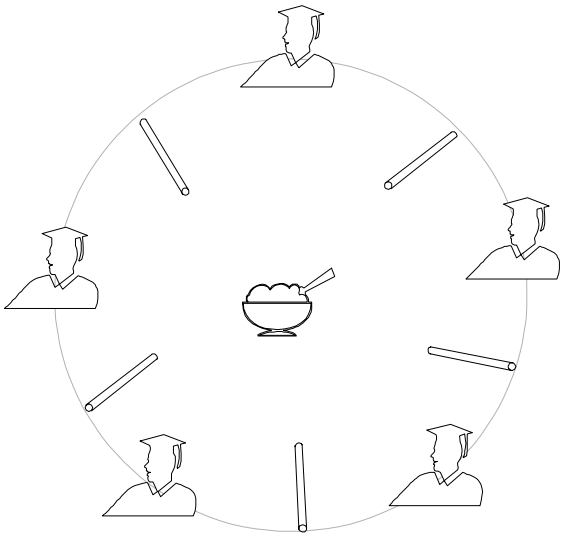


Figure 4.4.2 Picture representing the dining philosophers problem

This actually implements a precedence graph between the philosophers: a philosopher *Aristotle* has precedence over another philosopher *Plato* with respect to a chopstick *c* if, and only if, *Aristotle* holds chopstick *c* and *c* is clean, or *Plato* holds *c* and *c* is dirty, or *c* is being transferred from *Plato* to *Aristotle*. If the group of philosophers is seated at the table such that the precedence graph is acyclic, then it will remain acyclic because: precedence of *Aristotle* over *Plato* can only be reversed when *Aristotle* starts eating, at which time *Aristotle* precedes over both his neighbours, which implies that the graph is acyclic.

We will discuss an extension to this synchronisation problem called the 'evolving philosophers problem', which was proposed in [Kramer 90]. In this extension the configuration of philosophers can be modified, in that philosophers can be removed from (*death*) and added to (*birth*) the ring configuration. This extension is used to demonstrate two properties of the method: its support for specialising and extending objects and the support for dynamic object creation and deletion.

The extension requires the philosophers to go into a passive state, and be activated again later. This is necessary because during the birth or death of a philosopher the configuration of chopsticks must be rearranged such that an acyclic precedence graph is retained. During this rearrangement the current configuration should not change, as this could cause inconsistencies. We will focus on the synchronisation of the philosophers only, for a detailed discussion of the algorithm for rearranging chopsticks we refer to [Kramer 90].

4.4.3 Step I. Object Identification

We first give the template description for this step:

Step I: Object Identification.
Input: - (requirements specification and subsystem context)
Output: a set of objects.
Side-effects: -
Description: by looking at the problem domain and requirements specification, relevant and stable entities are identified.
Hints:
- ❑ look at the real-world.
- ❑ activities or processes may be modelled as objects.
- ❑ interaction patterns can be abstracted to objects.

Substeps:
- I.1 Candidate object identification.
- I.2 Candidate object elimination & merging.
- I.3 Object multiplicity definition.

Description: All the method steps in this section deal with a single subsystem. Obviously, at the initial step of the analysis, no subsystem may be defined yet. Thus, one of the tasks in this step is to define the boundaries of the subsystem. At a later phase or during an iteration cycle, a subsystem where (additional) objects are to be identified may be given. In this case the boundaries of the system are largely fixed.

The object identification phase bears some resemblance with a brain-storm session: the first stage is to collect as many candidate objects as possible. This should also provoke some thoughts about the nature of the subsystem: at the second stage, candidate objects that are not adequate are eliminated. In the third stage the multiplicity of the objects that are left is dealt with.

Note that for all steps in the method, the requirements specification and the subsystem, including its context, is assumed to be available as input. Thus, in the forthcoming discussion of method steps these will not be mentioned explicitly.

Step I.1 Candidate Object Identification.

Step I.1: Candidate object identification.
Input: -
Output: a list of candidate objects
Side-effects: -
Description: This is a first attempt to the definition of a proper set of objects. It can be compared to a brain-storm session, where any proposed object is registered. In later (sub-)steps redundant and inappropriate objects will be removed from the list of objects.

Hints:
- ❏ Look at the real-world (this *may* be the computer science domain).
- ❏ Activities or processes can be modelled as objects.
- ❏ Interaction patterns can be abstracted to objects.

Description: Finding candidate objects is based mainly on the requirements specifications and the context of the subsystem (when available). In the text of the requirements specification potentially suitable entities can be found. Not only nouns may point towards suitable candidates (as proposed in [Abott 83]), but verbs, activities or processes may indicate potential objects as well. The candidate objects are compiled into a list that will be processed in subsequent steps.

Hints: It is quite important to construct an object model that closely resembles the real-world domain. Note that for technical problems in the computer science domain (for example operating system services), entities within the computer system may become 'real-world': resources, devices, memory, schedulers, etcetera. The same is true when this step is performed during design: the objects that are to be identified may include such things as linked lists, caches, files etcetera. Especially resources are very suitable to be modelled as, or encapsulated by, objects. This allows access to the resources to be controlled and synchronised and organised through inheritance hierarchies.

In addition, the actors in the problem domain can also be identified as objects. Note that certain interactions or communication patterns are also candidate objects. More information on finding (and eliminating) candidate objects can be found in e.g. [Coad 91a, 91b] and [Whiting 90].

Step I.2 Candidate Object Elimination.

Step I.2: Candidate object elimination.
Input: a list of candidate objects.
Output: lists of accepted and eliminated objects.
Side-effects: -
Description: all candidate objects are examined, and either accepted or eliminated.
Hints:
- ❏ Does the object lie within the system boundaries?
- ❏ Look at organisational, structural and interaction relations.
- ❏ Consider functional and storage behaviour.
- ❏ Merge candidate objects that represent the same entity.

Description: The list of candidate objects is scanned, and for each candidate object it is decided whether it is a relevant object, or should be eliminated from the list of objects. The following hints may well contain relevant criteria for deciding upon this:

Hints: Firstly, each object should have its own, unique, identity: in case multiple objects represent different views of an entity with a single identity, they should be merged. Objects should play a significant role in the system under consideration: by representing some information that is to be maintained or remembered, or by offering certain services. In

addition, they should have some relation to the other objects in the system: there should be either some form of communication, or a structural relation with other objects. Such decisions may be taken only after the method steps III and IV (usually during an iteration phase).

Step I.3 Object Multiplicity Definition.

Step I.3: Object multiplicity definition.
Input: list of objects.
Output: list of objects and collection objects.
Side-effects: -
Description: For all the objects that have been identified so far, multiplicity is considered, related to the subsystem context.

Description: The approach of the method towards objects versus classes is rather dual: we emphasise regarding objects in object diagrams as unique instances rather than the abstract notion of classes. On the other hand these object specifications are mapped straightly to class definitions. The reason for putting emphasis on objects as instances is that it allows for considering them in configurations of objects, which is more proper to reveal the interaction between objects. The interaction between objects is crucial when considering the synchronisation of objects.

When considering object configurations, the relative multiplicity of objects is very important: whether an object has to serve requests from a single or from multiple clients can heavily influence the synchronisation requirements. Therefore, for each proposed object in the subsystem, it must be defined whether, in a single instantiation of the subsystem[2], a single object exists, or a collection with multiple objects.

The example:
For the dining philosophers problem a large number of -candidate- objects can be identified, but to model the synchronisation problem, we need only a small subset of these. Some candidates that can be derived from the problem description are: philosophers, chopsticks, ricebowl, rice, hands (holding chopsticks), neighbours, table, precedence graph, cycle, group of philosophers, clean/dirty and think/eat/getHungry. This includes all types of words mentioned in the problem description, including nouns and verbs. From these candidates, we selected the relevant objects, and combined as shown in figure 4.4.3.

It should be mentioned explicitly that for every problem, a large number of solutions exists, including several 'good' solutions[3]. The solution that will be proposed here does not rule out other possibilities. In this solution we try to simulate the 'real-world' of the dining

---

2 Recall that we can talk about subsystems as if they are objects, because they *are* first-class objects.

3 The judgement whether a design is 'good' or 'bad' fully depends on the requirements: a lot of design requirements conflict with each other, and it must thus be made explicit which properties are considered to be the most important ones. In this thesis, we concentrate on reusability, extensibility and maintainability.

philosophers problem. We model the philosophers as objects that communicate by sending messages to each other (to their left and right neighbours, to be exact). Thus, the responsibility for synchronising their actions is put at the philosopher objects.

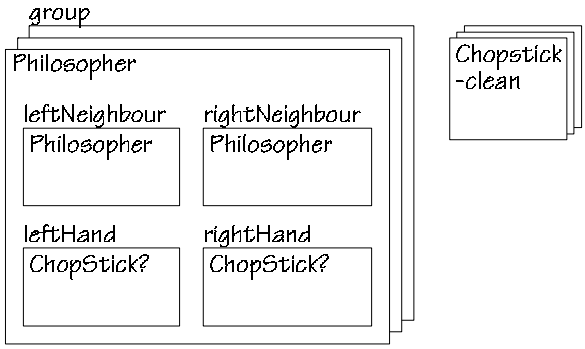


Figure 4.4.3 The selected objects

The relevant objects in the system thus are the philosopher and the chopsticks. The chopsticks are relevant objects for two reasons: they need to be exchanged between neighbouring philosophers and they include a state (they can be dirty or clean). The picture already shows some of the structure of the system: as each of the candidate objects is considered, their role in the system can be visualised immediately in an object diagram.

The philosopher and chopstick are the only two new classes in the system, however a number of candidate objects appear in the picture in different roles: every philosopher indeed has a left hand and right hand in the form of parts that can 'hold' (or refer to) a chopstick. Analogously, every philosopher refers to its left and right neighbour. In addition, we defined that the system contains multiple philosophers: this collection is termed group. We now covered the candidate objects philosophers, chopsticks, hands, neighbours and group of philosophers.

The other candidate objects are not considered to be relevant objects here, for reasons that we discuss per candidate object: ricebowl, rice and table play no role in the dining philosophers, even though we could for instance model eating from the rice bowl by sending messages ('get some rice, using these two chopsticks') to a rice bowl object, we decided that this has no significance to our problem. The more abstract notions of precedence graph and cycle do not need to be represented explicitly in this problem: this is already done implicitly through the states of the philosophers and chopsticks. The candidates clean and dirty are not interesting as full-blown objects here; they are merely relevant as the states a chopstick can be in, but do not have an internal state themselves, and have no responsibilities to fulfil. The activities think, eat and getHungry are simple commands, rather than complete objects: no internal state needs to be maintained for them, and no complex behaviour is expected.

Both the philosopher and the chopstick objects are defined to appear in the subsystem as collections. The reason is that a single ring configuration of philosophers requires multiple objects and multiple chopsticks. Assume, for example that we would like to model the rice bowl as well: this would appear as a single object in the diagram. Note that the definition of multiplicity is not a property of the object itself, but rather of the subsystem.

4.4.4 Step II. Specifying Objects

Step II: Specifying Objects.
Input: a list of objects.
Output: a basic object diagram (BOD).
Side-effects: identification of new objects.
Description: In this step BODs are drawn, defining properties of objects such as attributes, nested parts and object interfaces.
Hints:
- ❑ look at the responsibilities of the object.
- ❑ look at the nested objects.
- ❑ look at the context (subsystem & objects).

Substeps:
- ❑ Object labels
- ❑ Class names
- ❑ Multiplicity of objects
- ❑ Interface of incoming and outgoing messages
- ❑ Attributes
- ❑ Parts

The basic object diagram notation was presented in section 4.2. The input for the diagram is a list of objects; this allows to start drawing a set of rectangles. We will now discuss the various aspects of objects that are defined in BODs. Some hints for finding these aspects are inserted at the relevant locations:

labels: each object in an object diagram is tagged with a -for that subsystem- unique identifier. Although these labels do not play an essential role, they are required during the realisation phase[4], and may serve a better understanding of the system. Therefore, we suggest to label all objects in the diagram immediately.

class names: as each object is an instance of a class, the name of its class must be specified. The purpose of this may be two-fold: the class name can be perceived of as a reference to the definition of the object elsewhere. On the other hand, it can be used as a declaration: the class is defined by the properties of the object.

multiplicity: in step I.4 multiplicity of objects was derived, this is revealed by the object diagram notation as well.

interface: The interface of an object consists of two aspects: the first is the input interface, which defines the set of messages that the object accepts. This interface is largely determined by the *responsibilities* of the object[5]. Thus, messages on the input interface can be found by looking at the services and functionality that are expected to be offered by the object. Or by looking at the attributes and parts: does the object need to provide access to -some of- its nested objects? The responsibilities of the subsystem and the

4 Unique labels can be generated automatically, but this does not contribute to our goal of constructing well-formed and (re-) usable objects. For rapid prototyping it could be acceptable, though.

5 This is essentially a specification of what an object should do: see e.g. [Wirfs-Brock 90a].

other objects in the subsystem may also give some clues about the messages that are to be supported.

The second part of the interface is the output interface: the messages that are sent by the object. In general, these messages will somehow serve to implement the responsibilities of the object. Thus, a look at the expected behaviour, and at the behaviour of the nested objects may reveal aspects of the output interface.

Another technique for determining the interface of an object is through *scenarios, walk-throughs* or *use-cases*. We will briefly discuss these when looking at object interactions.

attributes: the attributes of an object are, just like parts, nested (first-class) objects. The only distinction is that nested objects which are reused on the interface of the current object are always drawn as part objects instead of attributes. In general nested objects are denoted as attributes for layout purposes and convenient drawing, and if the object is an instance of an existing, well-defined class.

parts: nested objects are drawn as parts if: the object is not very well defined yet, the nested object is reused on the interface of the encapsulating class, or the nested object is relevant for defining the encapsulating object, e.g. for synchronisation or message interactions.

The method supports the definition of parts and attributes in this phase of the development, but it is only discussed explicitly how to find these in the section on object interactions. At the current method step, it is attempted to characterise the objects that have been identified so far as much as possible. As the attributes and parts play an important role in the characterisation of objects, they are included here.

The example:

The following picture shows the BOD for the dining philosophers problem:

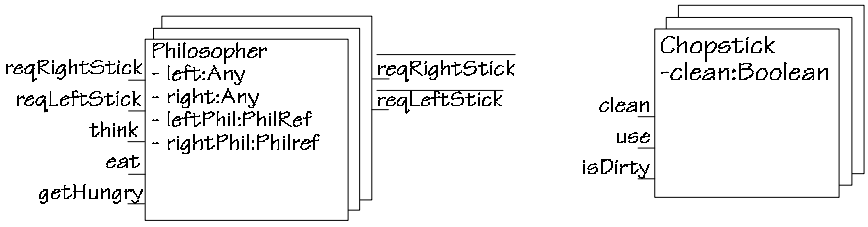


Figure 4.4.4 The BOD of the philosophers problem.

In the BOD the object description contains the external interface of the object as well. To make the picture smaller, we used the attribute notation for the nested parts. The left and right attributes are defined to be of type Any, as they will hold either a Chopstick object or a Nil object. For the leftPhil and rightPhil we just state that they are references to other Philosopher objects: this can be indices in the collection, pointers, unique identifiers, etc. We do not go into this level of detail.

The methods that are identified at this point are those for requesting chopsticks, and the methods that cause the state changes of the philosopher: from thinking to hungry, from hungry to eating, and from eating to thinking again.

4.4.5 Step III. Define Structural Relations

Step III: Define Structural Relations.
Input: Object Diagram.
Output: Object Structure Diagram.
Side-effects: creation & elimination of objects, restructuring BODS.
Description: This step deals with the identification and definition of the structural relations. These are: inheritance, delegation and aggregation.
Hints:

- ❑ Model real-world relations only.
- ❑ Search for kind-of hierarchies, e.g. taxonomies for the application domain.
- ❑ Reuse analysis and design models.
- ❑ Sharing behaviour with state.
- ❑ Resource encapsulation.

Description: An object-oriented application is structured along two[6] orthogonal axes: *sharing* relations and *part-of* relations. The first axis is formed by the sharing relations, which organise the objects in the system through generalisation-specialisation relations (see also [Booch 90]). The sharing axis is formed by two types of relations between objects: inheritance and delegation. In general, inheritance contributes most to a clear organisation and structuring of the system. It can be characterised as a kind-of relation. Delegation (as may also be the case for inheritance) focuses more on sharing of services.

Some hints on identifying sharing relations:

Kind-of hierarchies: one of the aims of the object oriented analysis phase is to structure the application model though a kind-of hierarchy, which may actually take the form of a lattice, in case an object has sharing relations with multiple ancestors. The main goal of such a hierarchy is to structure objects by considering specialisation-generalisation relationships between objects: the most general objects are seated at the top of the lattice, whereas the most specific objects are at the bottom of the lattice. The application model may contain several such lattices.

Taxonomies: As a part of the analysis process, some research into the problem domain is required. This may reveal existing theories and knowledge, that is often structured in the form of classification hierarchies or taxonomies. Such domain knowledge can be adopted for structuring the objects through inheritance relations.

Reuse: one of the frequently emphasised issues in object-oriented design is the notion of reuse. Sharing relations like inheritance and delegation are very suitable to express this. This is true for the reuse of code in the implementation phase, but also for reusing an

6 This -seemingly- ignores the structuring provided by subsystems. Firstly, because this is in general only available as a tool for the analysis and design phases. Secondly, because in our case, we use aggregation relations to provide sub-system functionality, which is actually the second axis described here.

analysis model. If the developer has access to previously developed models or libraries of models, a sharing relation to suitable pre-existing objects can be made. Extension and redefinition may tailor the reused objects to match the properties that are required for the system under development[7].

Sharing behaviour with state: the services that are offered by an object ('server') usually require some internal state to be maintained. This may reveal itself in two forms to objects that share the service ('clients'): for each object that shares a service a separate state must be maintained. In this case an inheritance relation is appropriate. It may also be the case that a single state must be maintained even though multiple objects share the service. In this case, both the behaviour (the services offered through methods) and the state (the nested objects) are to be shared, which requires a delegation relation from the client to the server object.

The second axis is the part-of structure of objects. This relation appears in many forms in the real-world: for instance as physical containment, organisational structures and conceptual grouping. The part-of relation also has the property of hiding the parts. This property can be applied for the encapsulation of resources. This allows for controlling unauthorised access, managing synchronisation, scheduling requests, etcetera. Part-of relations may also be derived by considering the -part- objects that could be required to provide the services of the -whole- object.

With all the suggestions above, it should be kept in mind that the ultimate goal in both analysis and design is the proper modelling of the real-world relations: it has been observed that the 'invention' of relations may work out well in a given subsystem, but is likely to cause problems when it is attempted to extend or reuse -parts of- the subsystem[8].

The example:

The dining philosophers problem that we use as an example is quite simple with respect to the objects and their structural relations. We propose some extensions to the basic problem to demonstrate more interesting structural relations. The first extension is the requirement to provide a more complete interface to make philosophers behave and react as human beings. Assuming that the modelling of human beings has been done before (e.g. for other parts of the application containing this subsystem), we let the philosophers inherit from Person. Class Person features two attributes, id and sex, and has an aggregation relation defining two part objects; one describing an address, and one describing the physical properties of a person.

The second extension is to make the chopsticks part of a hierarchy of (pre-defined) tool classes: Chopstick is a specialisation of Cutlery, which in return is a specialisation of Tool.

---

7 Reuse during the analysis phase has been largely ignored in the literature. This is possibly due to the risk that 'easy' reuse of pre-existing object specifications leads to improper modelling. For the realisation of a 'software factory', however, the use of libraries of -analysis- components for a certain domain is a must.

8 In fact, the combination of expressive power with strong encapsulation of composition provided by the composition-filters model greatly reduces this risk, compared to e.g. a model that supports single inheritance only. We still advise prudential utilisation of structural relations.

In the figure we added classes Hammer and Fork to indicate how the specialisation-generalisation tree may extend in its width:

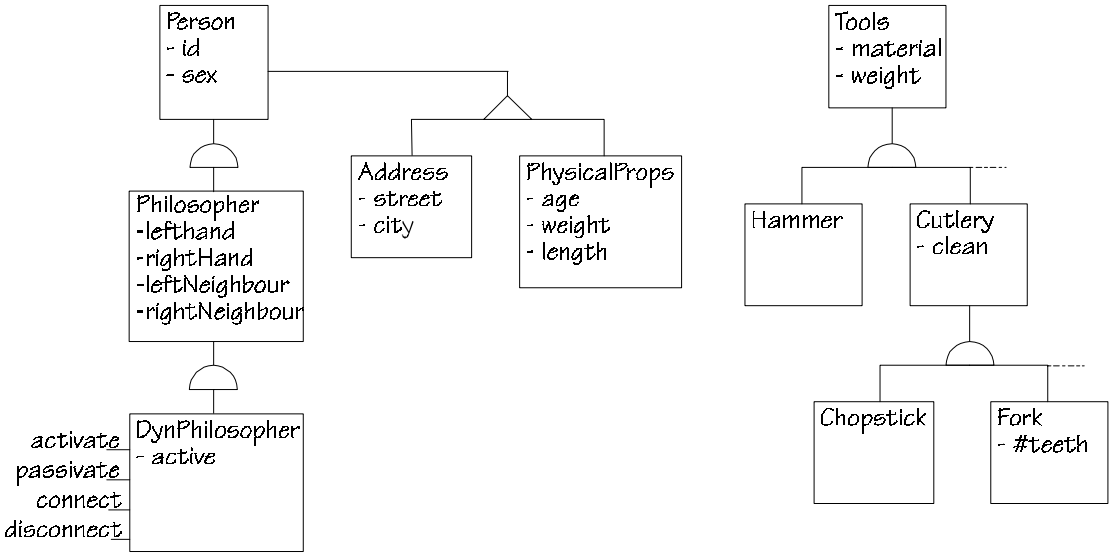


Figure 4.4.5 The OSD, including some optional extensions.

The last structural aspect that is visible in the OSD is the introduction of two kinds of philosopher objects: Philosopher and DynPhilosopher. The latter is an extension of the plain philosophers, and allows for dynamically adding and removing philosophers from the cyclic configuration of philosophers. This requires the methods connect, and diconnect, which make new links to neighbouring philosophers[9]. For a proper rearrangement of the chopsticks, the philosophers must be made passive, and active again, with the passivate and activate methods, respectively.

4.4.6 Step IV. Define Object Interactions

Step IV: Define Object Interactions.
Input: Object Diagram.
Output: Object Interaction Diagram.
Side-effects: extension of the interface of objects.
Description: The message interchange between objects in a typical configuration is defined. This reveals aspects of the dynamic behaviour of objects and details their interfaces.

Hints:
- ❑ look at the interfaces of the objects.
- ❑ use scenarios.
- ❑ activities that are to be synchronised.
- ❑ access to shared resources.

9 Actually, such methods are required as well in the static case for initialisation purposes. The real extension in DynPhilosopher is the additional synchronisation that is required.

Description: Object Interaction Diagrams (OIDs) define so-called *message connections* between objects. The existence of a message connection between two objects means that a certain set of messages is exchanged between the objects during their life-cycle. However, it does not give any clues about the pattern of these interactions[10]. We do not pay attention here to a more detailed definition of the messages, such as the specification of arguments and return types. This is only because it is not our primary interest, it should not be ignored however in a more general development method.

Hints:

- By looking at the interfaces of the objects, we see which messages they expect (input interface) and which messages are sent (output interface). This will give important clues as to which messages are interchanged between objects.
- Use scenarios (or walk-throughs, use cases) to investigate the message interactions. A scenario can be seen as a test-drive: for the typical services of the system under consideration it is observed which messages are sent to which objects, and what messages are triggered as a result. This is especially useful as a verification of the completeness of message connections and object interfaces that have been identified so far. More extensive discussions of scenarios can be found in [Wirfs-Brock 90] and [Rumbaugh 91].
- A message is not only a means of exchanging information, or a request for some service, it can also be a means for synchronising activities. Messages can be used to synchronise activities in different objects, but it is very well possible to synchronise the activities *within* an object by sending messages to itself.
- The use of shared resources in a system implies that a number of client objects will send messages to the shared object. This may be to retrieve information from the resource (queries), the requests can be commands to affect the behaviour, or they may be updates of the state of the object.

The example:

We distinguish three types of interaction relations in the philosopher subsystem: firstly the messages that are sent by the philosopher objects to the chopstick objects (using the chopstick, cleaning it, and retrieving its state respectively). Secondly the messages that the philosopher objects sent to each other (requesting chopsticks from neighbours). And thirdly the messages that are to be sent in order to manage the configuration of philosophers. This includes the messages to change the state of the philosophers between thinking and eating, and the messages for configuring the set of philosophers in the case of DynPhilosopher. We

[10] On one hand, the interaction patterns, or protocols, between objects can often have a significant effect on the synchronisation requirements for objects. From this point-of-view, it would be relevant to specify the interaction patterns. On the other hand, we strive to develop objects that are as much self-contained and independent of their context, as this increases reusability and reduces maintenance efforts. Therefore we opted not to promote the specification of interaction patterns.

do not make explicit which object sends these messages, but will assume there is a 'scheduler'[11]. The resulting OID is as follows:

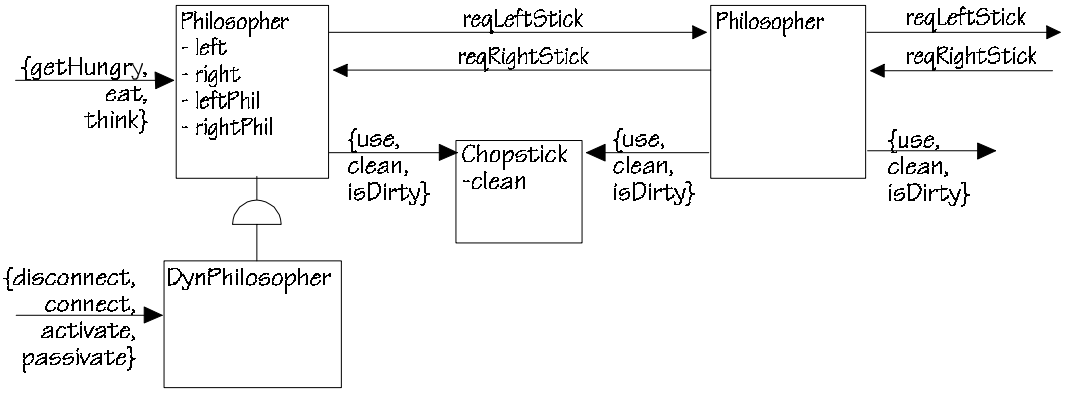


Figure 4.4.6 The object interactions in the dining philosophers problem.

4.4.7 Step V. Determine Synchronisation

Step V: Determine Synchronisation.
Input: Object diagram.
Output: SCD.
Side-effects: new sharing/reuse relations, new methods.
Description: In this step and its substeps the synchronisation of objects is determined. This is done on a per-object basis, where the context of the object, especially the reuse relations and message connections to other objects, are particularly important.
Hints:
- ❑ shared resources require synchronisation
- ❑ when in doubt, model the synchronisation
- ❑ complex interaction patterns

Substeps:
V.1 Reuse of SCDs
V.2 Decompose the Synchronisation Specification
V.3 Identifying states
V.4 Identifying events

Description: For all objects in the current subsystem the synchronisation is to be determined. Depending on the application domain, the majority of objects will not deal with synchronisation issues. These objects do enforce mutual exclusion, though: this will avoid inconsistencies in a concurrent environment.

For the objects that are likely to involve synchronisation, we identify two issues: the first is to allow intra-object concurrency (see step VI.3 for a discussion on intra-object concurrency). This means that the default synchronisation for objects, which realises mutual exclusion, must be discarded. The second is the scheduling of certain messages in order to

[11] There are several options: each philosopher could have its own internal scheduler, there could be a separate scheduler object in the subsystem, or the scheduler could be external to the subsystem.

avoid inconsistencies. Note that in the case of intra-object concurrency it is very likely that some scheduling of the received messages is required.

For each object that requires synchronisation, the substeps V.1 up to V.4 are to be performed, which results in an object with zero or more SCDS. The following figure visualises the contribution of each step to the specification of the object:

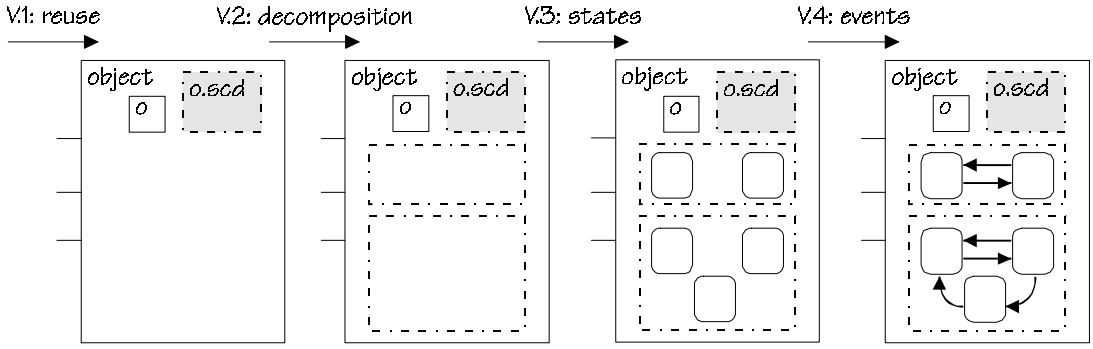


Figure 4.4.7 An overview of the substeps for the construction of SCDs.

As the figure shows, in substep V.1 it is tried to reuse SCDs from pre-existing objects (in the figure from an object o the SCD labelled scd is reused). Step V.2 decomposes the synchronisation, defining two new SCDs. In steps V.3 and V.4 these new SCDs are worked out, by defining respectively the states and the events of the SCD.

The ordering of these steps supports a top-down approach to the derivation of synchronisation specifications. This is merely intended to guide the developer through the initial phase. The steps are so much intertwined and interdependent that an incremental approach is inevitable.

Hints:

- Likely candidates for active objects[12] are objects that encapsulate shared resources, or instead objects that compete for shared resources with other objects. In the first case, synchronisation will concentrate on the incoming messages, whereas in the latter case synchronisation of outgoing messages is more likely to be required.
- In the cases where it is not clear whether synchronisation of messages is required, or not, it is advisable to follow the forthcoming steps: this will determine the appropriate synchronisation, which may turn out to be no synchronisation at all, or just mutual exclusion.
- Synchronisation constraints may be required because of the communication with other objects: to order and synchronise the respective activities. Complex interaction patterns are a clue that synchronisation may be necessary.

[12] Remember that an object is called 'active' when it can control its own synchronisation.

Step V.1 Reuse of SCDs

Step V.1: Reuse of SCDs.
Input: Full object diagrams.
Output: new reuse relations.
Side-effects: -
Description: We start defining synchronisation by trying to reuse synchronisation constraints from other objects.
Hints:
- ❑ adhere to naming conventions to facilitate reuse.
- ❑ follow the incremental specification approach.
- ❑ synchronisation constraints of a reused object can not be weakened.
- ❑ consider the composition of synchronisation specifications and methods that are reused independently.

Description: Reuse is a valuable technique for limiting the development effort, decreasing complexity and size of applications, and providing additional system structure through an inheritance/delegation hierarchy. In addition, it can result in more reliable systems: if the library of components is well-tested, the application that reuses objects can benefit from this.

Our method supports the reuse of synchronisation code (on which we focus) through inheritance and delegation. This requires a strong understanding of the desired synchronisation constraints, since the developer has to make a proper judgement as to the suitability of the reused synchronisation constraints.

Hints:

- ❑ Finding components to be reused is tough: apart from knowing the desired synchronisation behaviour, one must also know how and where -in a possibly large library- to find it. This problem, which is not restricted to the reuse of synchronisation code, is not yet resolved. The use of a common terminology when naming objects, methods and SCDs may reduce the problems.
- ❑ When reusing and extending active objects: focus on the methods that are locally added (i.e. in the subclass or delegating object). Our approach to synchronisation specification is incremental: only the new and changed requirements in a reusing object have to be taken into consideration. In SCDs the reused methods may well appear as virtual events, though.
- ❑ When reusing an object, and adding additional synchronisation constraints for the inherited methods: just redesign the synchronisation specification. One should be aware that the synchronisation constraints of the object we are reusing remain in effect, and can never be weakened. It is advisable to review the resulting specification and remove redundant constraints: try to 'program the difference'.
- ❑ It is possible to compose a synchronisation specification that is reused with methods that are newly defined, or reused from another object. Try to reuse complete SCDs, maybe

add another SCD locally to constrain that specification. It may be necessary to rename messages.

The example:
In the dining philosophers example we find no explicit reuse of SCDs. However, the philosophers are mutual exclusive. Although the reuse of mutual exclusion synchronisation is normally implicit, for the sake of illustrating SCD reuse we make it explicit now:

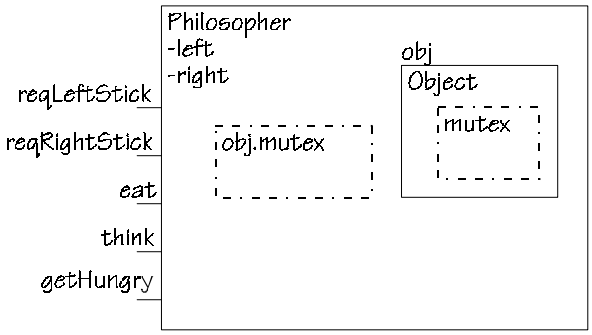


Figure 4.4.8 Reuse of SCDs: reusing the Mutex synchronisation from Object

Step V.2 Decompose the Synchronisation Specification

Step V.2 : Decompose the Synchronisation Specification. **Input:** Object diagram. **Output:** A set of SCDs. **Side-effects:** - **Description:** We try to decompose the synchronisation specification into several SCDs.
Hints: ❑ Find orthogonal life-cycles. ❑ Look at resources/nested objects. ❑ Look how the states or values of nested objects may affect synchronisation. ❑ Having similar states in an SCD may suggest splitting.

Description: In this step, we try to decompose the synchronisation specification into sub-specifications. This will give us a number of sub-problems that are most likely easier to model and solve than a single, complex specification.

The aim is to find a partition into smaller SCDs that are either completely independent, or must all be satisfied. This step can be performed recursively on the resulting sub-diagrams. Thus, it may be required to do this several times during step V.

Hints:

- ❑ Determine a set of orthogonal 'life-cycles': as an object is a composition of other objects and methods, it is often fairly easy to divide the SCD into two or more sub-SCDs that are independent of each other.
- ❑ Each subdiagram may focus on a particular (set of) resource(s), as represented by a set of objects. The synchronisation that is required may serve to protect these resources from inconsistencies. Inconsistencies can occur due to two situations: the first is

concurrent access by messages that modify the state of the resources, which is only possible when the object allows intra-object concurrency. The second situation is when the execution of a message in a particular state of the object will lead to an inconsistent state. For example, getting elements from an empty buffer, or a philosopher that grants a request for a chopstick while continuing eating.

- On the other hand, the state or value of objects may directly imply synchronisation constraints for messages. For instance, a Boolean attribute that indicates the sleep or active mode of the object, or a variable that can hold a token which enables certain operations.
- An indication for splitting an SCD into two or more SCDs is the -partial- replication of state specifications, or the occurrence of -groups of- states that share equivalent characteristics (including the outgoing events). In general, repetition of specifications or properties is a hint that splitting may be advantageous.
- Note that the nested objects may have their own synchronisation constraints. In some cases this fine-granularity may be advantageous as reusable abstractions or for further refinement later.

The example:

The first important SCD component that we can identify is one that defines the life-cycle of the object: each philosopher goes through a cycle of thinking, getting hungry (i.e. starting to collect the chopsticks from both sides), eating (once it has collected the chopsticks) and then thinking again. The stage within the life cycle affects the acceptance of requests for chopsticks: for instance, when eating, a philosopher will not give away its chopsticks. This SCD is labelled lifeCycle. The other relevant SCD components can be obtained by looking at the states that are made up by the parts of the objects: it is quite relevant that the chopsticks are really held before starting to eat. Thus we identify two additional SCDs that each model the availability of the left respectively the right chopstick: leftSync and rightSync. The motivation for distinguishing two separate SCDs for this is two-fold: firstly, it gives a more modular specification, and secondly, it concerns two separate constraints on the same message that must *both* be satisfied[13]. The resulting diagrams is as follows:

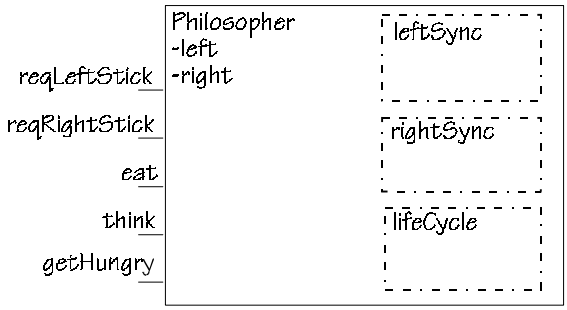


Figure 4.4.9 The components for the synchronisation of the dining philosophers.

[13] The reason for having distinguished methods for the left and right chopstick is purely practical: the alternative of having one method with a parameter that specifies whether it concerns the left or right chopstick gave slightly more complicated and less readable code.

Step V.3 State Identification & Specification

Step V.3: State Identification & Specification.
Input: SCD (this can be an empty template).
Output: SCD (with states).
Side-effects: Identification of new parts.
Description: The identification of new states for the SCD. This includes a full specification in terms of the object state.
Hints:
- ❑ Look at nested objects, external objects or the object manager.
- ❑ Find messages and events that may cause inconsistencies.
- ❑ Keep states within an SCD mutual exclusive.

Substeps:
- ❑ Identify a new state, define appropriate name.
- ❑ search for reuse of state specifications
- ❑ Specify the state condition.
- ❑ Specify the action annotation.

Description: In this step, (that is to be applied for each SCD), a set of states is identified and specified. The synchronisation of messages is a protection mechanism: it is meant to guard the object against undesirable situations and inconsistencies. An SCD presents the states an object can be in in an abstract manner. A state is relevant when it describes a situation where the acceptance of a particular message, or set of messages, may endanger the consistency of the system. On the other hand, a state may be relevant as well when it indicates a situation where messages *can* be accepted safely.

In this step, these 'safe' and 'dangerous' states are to be found and specified. The specification of a state is a message expression in terms of the values or states of nested objects, or the state of the object manager. Refer to chapter 3 for a detailed description of the object manager characteristics.

Hints:

- ❑ Look at attributes and parts of the object, as these make up the larger partition of the interesting states. However, it is of no interest to try to define a particular state for every configuration of instance variable values. States must have a purpose and meaning with respect to synchronisation problems.
- ❑ In some situations, the state of external objects may be relevant for the synchronisation.
- ❑ Look at the object-manager: in many situations, the state of the object manager, reflecting the number of active or queued requests, is relevant with respect to synchronisation problems.
- ❑ Look at the messages on the object interface and consider for each whether they might cause inconsistencies in particular situations. Then abstract these situations into an object state specification.
- ❑ For the purpose of clarity and reliability, it is advised to keep the states within a single SCD mutually exclusive. This means that within a single SCD, at each particular moment, only one state should be valid, or *true*. The method does not strictly demand this, neither are there means to check this; but this approach will result in SCDs that read like

finite state machines. The method promotes these as intuitive, readable abstractions of the dynamics of objects[14].

Substeps:

- ❑ Identify a new state, using the hints given above, and define an appropriate name.
- ❑ Search for possible reuse of the state specification; is there a similar state defined by any of the other SCDs, or by one of the nested objects? Do keep in mind that the reused state should not merely have a correct condition, but both states should be intended to express exactly the same situation. This is important in order to cope with future changes and system evolution.
- ❑ Specify a Boolean expression that characterises the state: when the expression evaluates to true, the object is said to be in the corresponding state, when the evaluation of the expression results in the value false, the object is not in that state. Note that the message expression should not involve messages that cause side-effects.
- ❑ (Optionally) specify the action annotation that is associated with the state. The action annotations are primarily intended as a documentation for the developer: it allows for the explicit specification of state changes and the flow of control. Action annotations can be expressed informally by natural language, or by (pseudo-)code.

The example:

For each of the three SCDs that were identified, the states that make up the life-cycle are drawn. At the same time, the events that cause the transition from one state to another are drawn: these are an important clue for an intuitive interpretation of the diagrams.

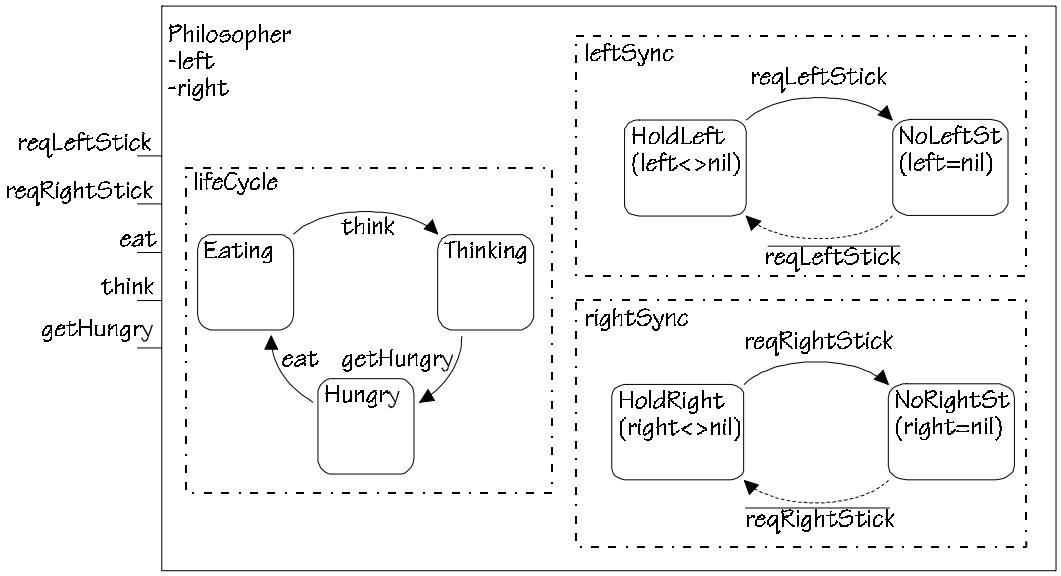


Figure 4.4.10 The definition of states, with the basic transitions that define the life-cycle.

The life-cycle SCD consists of three states: Thinking, Hungry and Eating. The transitions from one state to another are caused by the respective messages getHungry, eat and think. These transition messages are -happen to be- allowed only when the object is in the

[14] Even though FSMs aim at describing causal relations that we are not primarily interested in.

corresponding state (therefore they can be drawn as solid arcs, i.e. they are also synchronised). Note that the states are not specified by a Boolean expression: this indicates that each state is defined by a separate Boolean variable. The messages that cause the transitions must update these variables.

The leftSync and rightSync SCDs model the ownership of the left respectively right chopstick. the object is assumed to hold a chopstick when the left respectively right variable is not nil. The transitions between the HoldX and NoXSt states (where 'X' can be 'Left' or 'Right') are caused by the reqXStick messages. A received reqXStick message is only allowed to execute when the philosopher really holds the corresponding chopstick[15]. The transition back is drawn as a virtual transition, caused by sending a reqXStick to the neighbour philosopher. The transition is made virtual because we want to avoid unnecessary synchronisation, and we can be sure that the philosopher will only send a reqXSt message when the corresponding chopstick is not available.

For the DynPhilosopher class (we left out the reuse of mutex from Object) we define the following diagram:

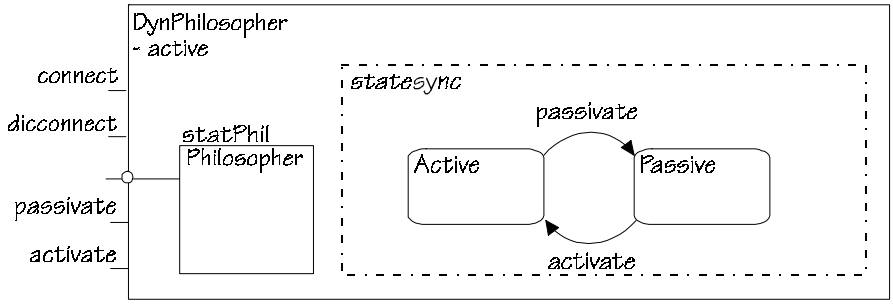


Figure 4.4.11 The states and basic transitions of the DynPhilosopher object.

The SCD models the following two states a philosopher can be in: Active or Passive. The implementation of these states is again to be generated with the introduction of one or two Boolean instance variables. The two methods passivate and activate take the object from one state to the other (Thus they must modify the corresponding instance variables).

Step V.4 Identification and Specification of Events

Step V.4: Identification and specification of events.
Input: SCD (not including all events).
Output: SCD (possibly complete).
Side-effects: Identification of new methods, new message connections.
Description: The second component of SCDs, events, are identified in this step. Properties such as conditions on events and action annotations are to be defined as well.

15 Actually, this constraint is not necessary, as the neighbour should never ask for a chopstick when the philosopher does not hold it, because this implies that the neighbour itself holds it. But we do not want to depend on assumptions about the behaviour of the clients of an object.

Hints:
❑ Only deal with relevant events.
❑ Virtual events may be derived from the object manager state. ❑ Distinguish received events from generated events.
❑ Specifying read- and write-behaviour of events may reveal possible inconsistencies.
Substeps:
❑ Identify events. ❑ Distinguish virtual events from real events.
❑ Specify conditions on events ❑ Define action annotations on events.
❑ (Optionally) define early or late forks on events.

Description: Events model the transition between states. An event is actually a message activation: for incoming messages this means that the messages are allowed to be executed. For outgoing messages this means that the messages are sent to their respective receiver object. Events in SCDs have two purposes. The main purpose is the specification of which message is allowed to execute in a certain state. This is what synchronisation is all about.

The second purpose is to indicate the transitions from one state to another. It essentially does not matter what the target state of a transition is: this information has no influence on the synchronisation specifications derived from the SCD. The rationale for modelling the transitions completely is that this is expected to have a positive influence on the intuitiveness and readability of SCDs.

All events that appear in an SCD must be defined at the interface of the corresponding object. We distinguish several kinds of events: *received* respectively *generated* events, and *virtual* versus *real* events. These will be discussed below.

Hints:

- ❑ Each message that is accepted by an object, is an event by definition. However, for our purposes, we are only interested in those (message) events that are to be synchronised or cause relevant state changes.
- ❑ To find virtual events, look at the states that are expressed in terms of the state of the object manager.
- ❑ We distinguish *received* events, i.e. received messages, from *generated* events, i.e. outgoing messages, sent by the object at hand. Synchronisation will commonly occur for received events mainly, synchronisation of generated events is especially useful when the target object for some reason does not provide synchronisation. This can be because it is a pre-existing object, or because the object should not actively deal with synchronisation of messages for modelling reasons.
- ❑ For some applications it may be advantageous to specify the read- and write-behaviour of all events, as this is a good clue for finding data consistency problems and resource sharing conflicts. For generated events, the target object of the event should be specified.

SubSteps:

- Look at possible inconsistencies, from these derive the relevant events (i.e. primarily those that need to be synchronised). Inconsistencies are largely due to problems with resource sharing, and resources are represented by objects. Thus, look for objects (either instance variables, internals or externals) that represent some shared resource, and find the events that may conflict with each other. The latter is facilitated by looking at the read/write behaviour of events. Try to characterise the situations where no conflict will occur (and sometimes the situations where they *do* occur as well) and make sure the events leave only 'safe' states.
- Specialise the events into *virtual* events and *real* events: virtual events are those events that are not synchronised, but do cause state transitions. They merely serve schematic convenience, promoting the insight into the object life cycle by making a more complete diagram. Changes in the object manager state can only be modelled as virtual events.
- Specify conditions on events. Note that there may be more than one occurrence of an event in an SCD, and in other SCDs of the same object. *All* the occurrences of an event for an object have by definition the same conditions! A condition *C* can be replaced by a separate SCD that allows the transition only when the object is in a state S_C where *C* holds. The latter representation of conditions is preferred as it is less error-prone.
- When this is desired, associate actions (these are annotations only) with event occurrences. Similar to the association of conditions with events, an action annotation with an event should be replicated for each occurrence of the event in the object.
- Annotate transitions with the notion of early or late forks, when applicable. Be aware that these -especially the early fork- may cause inconsistencies due to concurrency.

The example:

The following figure shows the full object diagram of the Philosopher class:

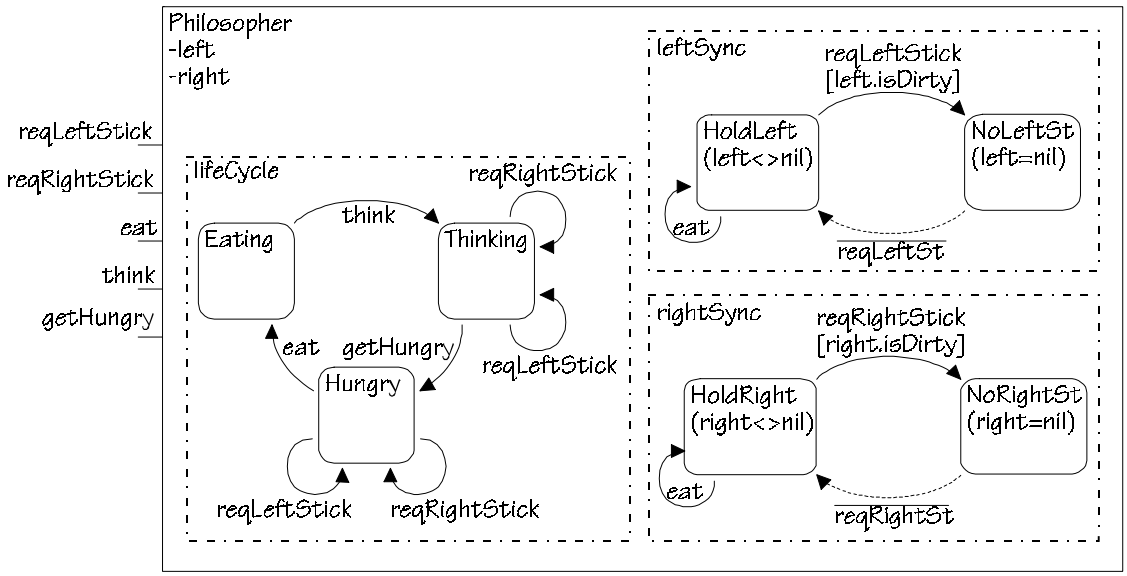


Figure 4.4.12 The complete SCD for the Philosopher object.

The Chopstick object will not be discussed, as it does not synchronise incoming messages. This is in accordance with the real-world modelling approach that we promote: a chopstick

is a purely passive object. The only methods that are provided on its interface are those that affect its state.

This is different for the Philosopher object; this can actively affect the received messages. The added events are mainly the messages that need to be synchronised. The lifeCycle SCD defines synchronisation constraints for the messages reqLeftStick and reqRightStick; only when a philosopher is not thinking or hungry, such a message is allowed.

The leftSync and rightSync SCDs impose additional constraints on the reqLeftStick and reqRightStick messages: a request for a chopstick is only accepted when the philosopher really holds it *and* (the additional condition written in square brackets) the chopstick is dirty. Another important constraint is specified by these two SCDs: the eat message is only accepted when the object is in both the HoldRight state (in the rightSync SCD) and the HoldLeft state (in the LeftSync SCD). Because the eat message appears in both these SCDs, the corresponding constraints must all be satisfied. The eat message does not appear in the lifeCycle SCD, however: this means that this SCD does not impose any synchronisation constraints on eat messages.

The synchronisation for DynPhilosopher is described in the following figure:

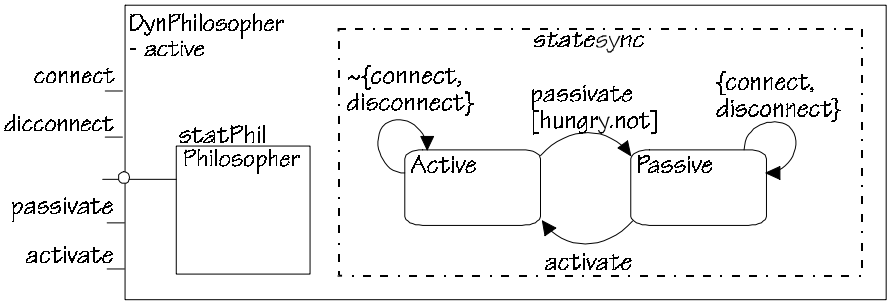


Figure 4.4.13 The complete synchronisation of DynPhilosopher, including all events.

The most significant feature of this diagram is the use of the exclusion operator '~'. The event specification '~{connect, disconnect}' means that all messages except connect and disconnect are allowed in the Active state. The result is that the diagram specifies constraints for *all* messages that arrive on the interface of the DynPhilosopher object, including the inherited methods. Thus, when the object is in the Passive state, only the connect, disconnect and activate messages are accepted. Transition to the Passive state is only allowed when the object is not yet in the Passive state and not in the Hungry state, where Hungry is defined by the nested statPhil object (informally: by the superclass).

4.4.8 Step VI. Iterating with Design Considerations

Step VI.: Iterating with Design Considerations.
Input: Full Object Diagram
Output: (modified) Full Object Diagram.
Side-effects: -
Description: The current configuration of objects is reconsidered, now focusing on typical design aspects.

Hints:
- ❑ Extensive SCD definitions for one object may suggest object splitting

Substeps:
6.1 Open-endedness.
6.2 Full specification of interfaces.
6.3 Sequential vs. parallel objects.

Description: This step tries to look at the SCD that was obtained from the previous steps not from a modelling perspective, but rather from the design perspective: focusing on issues like reusability, extensibility and maintainability.

Obviously, it is better to take the design issues discussed here into consideration even during the first iteration. On the other hand, first the modelling aspects should be concentrated on. In addition, we promote iterative development as an effective technique for creating reliable, well structured software. The re-consideration of a system with respect to design considerations only fits well into this philosophy.

Hints:

- ❑ When a single object contains a lot of SCDs, this may be an indication for splitting the object into separate parts. This should not break up the SCDs, but group the SCDs that share common events.

Step VI.1 Open-endedness

Step VI.1: Open-endedness.
Input: SCD.
Output: SCD.
Side-effects: -
Description: Reconsider the synchronisation specifications with respect to open-endedness, making the specification useful in extended contexts as well.
Substeps:
- ❑ Synchronise-set only.
- ❑ Synchronise-for-the-Future.

Description: This step focuses on ensuring the open-endedness of the described synchronisation: open-endedness benefits from writing a synchronisation specification such that it does not fully prescribe synchronisation constraints for a fixed set of messages, but takes into account that in the future the specification may be used in a context where additional messages appear. The key to open-endedness is the appropriate use of wild cards and exclusion.

Substeps: There are two basic approaches to describing open-ended synchronisation specifications. In either case a distinction is made between a fixed set of 'known' messages, and an open-ended set of unknown -or irrelevant- messages. The latter may change when

new messages are added to the object or to one of its superclasses, and when the specification (i.e. the SCD) is reused in another context:

- *Synchronise Set Only*: This approach imposes synchronisation constraints on the fixed set of known messages, and leaves all other messages undisturbed (i.e. imposes no synchronisation restrictions on these). This corresponds to the idea of 'minimal restriction'. A normal SCD without wild cards and exclusive transitions realises this approach: all messages that is not dealt with explicitly in an SCD are assumed to be unconstrained.
- *Synchronise-for-the-Future*: The second approach assumes that there are two kinds of messages: a fixed set of messages that each require individual synchronisation constraints, whereas upon all other -including yet unknown- messages another synchronisation constraint is imposed. This means that in the latter case, the synchronisation constraints are generic, and may be applied to messages that are not in the picture at the time of specification. This kind of specification is achieved through the use of wild-cards: the events on particular transitions can be replaced with a wild card (denoted by an asterisk). Sometimes exclusion of certain events may be necessary.

Actually, the first approach is a special case of the second, where the generic constraint is replaced by an empty, or True, constraint.

The example:

If we consider for example the leftSync SCD defined by Philosopher, we see that it defines synchronisation for a limited set of explicitly specified messages: eat and reqLeftStick. Other messages that are received by the object are not restricted by this diagram: the 'synchronise set only' situation. When this synchronisation specification is reused in another context, it will still only constrain the two explicitly specified messages.

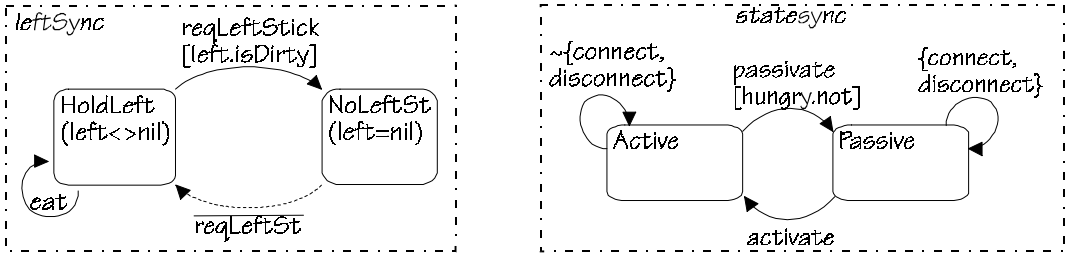


Figure 4.4.1 The leftSync SCD and the stateSync SCD from the Philosopher object.

The stateSync SCD on the right hand side of the figure is an example of the synchronise-for-the-future approach: due to the use of the '~' operator (wild card with exclusion), the diagram affects all the messages that are received by the object. When this synchronisation specification is reused in another context with other messages, or the object is extended with new methods, the constraint will also apply to these new messages.

Step VI.2 Full Specification of Interface

Step VI.2: Full Specification of Interface.
Input: SCD.
Output: SCD.
Side-effects: -

Description: For optimal reuse of -reused- synchronisation specifications in subclasses, reused components must explicitly be made available on the interface of the object.
Hints:
- ❑ offer an abstraction of the internal object state at the interface.
- ❑ when the object state is accessible through methods most synchronisation problems can be resolved.

Description: An important requirement for reusable software is that the interfaces of objects are maintained properly (see also [Matsuoka 93b]). This is a slightly complicated issue, as there is a trade-off between encapsulation and reuse. The composition-filters model simulates inheritance by redirecting messages through the dispatching mechanism. Thus, a 'subclass' puts *only* the inherited methods on its interface (as is specified by reuse relations in SCDs, and by the Dispatch filter in a composition-filters application). However, for optimal reuse of synchronisation specifications, complete SCDs and single states specifications (cf. filter specifications and condition implementations) must be available for reuse.

The key issue here is that the object is responsible for putting the SCDs and states it reuses on the interface, even though this partly reveals its internal (reuse) structure. This is true both for inheritance and delegation.

> There is a language design trade-off here: weakening of encapsulation so that filters and conditions visibility is a transitive property over the subclassing hierarchy would solve the reusability problems. However, apart from the fact that this does not fit in well with composition-filters approach to reuse, it is generally recognised that this is not a good thing to do from the software engineering perspective [Snyder 86]. Therefore, the application programmer is responsible for making inherited SCDs and states available to its 'subclasses' (i.e. those objects that include the object as an external or internal).

Hints:

- ❑ One should aim at offering an abstraction of the internal state of the object at the interface. This must be abstract enough as to hide implementation aspects, and sufficiently expressive so that subclasses can implement synchronisation based on the state of the inherited object.
- ❑ When the internal state of the (reused) object is accessible through methods, the need for putting the conditions and filter specifications on the interface becomes less urgent: this allows the (re-)implementation of conditions and filter specifications. This is more flexible, but it is less abstract and may require extensive redefinition of conditions and filter specifications.

The example:

The issue of maintaining interfaces can be appropriately illustrated using the dining philosophers example: firstly by the interface that the Philosopher object must offer so that the DynPhilosopher subclass can be implemented: the neighbours are encapsulated by the Philosopher class, but must be accessed by the connect and disconnect methods defined in the DynPhilosopher class. Thus, the interface of Philosopher must be extended with methods for reading and writing the left and right neighbour.

A second interface issue is that DynPhilosopher refers to the state Hungry that is defined by Philosopher; states are always accessible to their direct descendants, but now DynPhilosopher has the responsibility to make the relevant states of Philosopher available to its own descendants by putting them on the interface.

Step VI.3 Sequential vs. Parallel Objects

Step VI.3: Sequential vs. Parallel Objects.
Input: Full object diagram.
Output: Full object diagram.
Side-effects: Intra-object concurrency, additional synchronisation.
Description: Reconsider for which objects internal concurrency is relevant, and which should remain mutual exclusive (sequential).
Hints:
- ❑ Creation of additional concurrency.
- ❑ Avoid externally visible side-effects after an early return.
- ❑ Intra-object concurrency increases the amount of concurrency.
- ❑ Watch for inconsistencies in case of intra-object concurrency.
- ❑ Beware of creating concurrency only for increasing performance.

Description: Concurrency, or parallelism, and synchronisation are interdependent: if a system supports concurrency then some synchronisation is required, but the reverse is true as well: synchronisation without any concurrency is useless, if not impossible. Consider for example the bounded buffer example: blocking *get* or *put* messages in a program with only a single thread causes dead-lock. Therefore we pay attention here to the creation and preservation of concurrency in the system.

Intra-object concurrency is especially important because we promote systems that are strongly hierarchically structured through encapsulation and scope rules. This is in contrast with the approach of a so-called 'sea of objects' where all objects drift in a global scope, and can all directly access each other without passing -multiple- object boundaries. In addition, we promote mutual exclusive objects as the default strategy for fighting inconsistencies. This has the effect of serialising the activities in the system.

Hints:

- ❑ There are two ways for creating new concurrent activities. The first is through the mechanism of early returns in method implementations. This is also dealt with in the form of early and late forks in step V.4. The second way of creating additional concurrency is by initial processes: whenever a new object is created, a new thread is started to execute the initial method. It is common that the initial method performs some initialisation of the object only, and then terminates. But the initial method may well send messages to other objects, and could remain active throughout the life-time of the object[1].

[1] To achieve this, an early return statement must be issued by the initial method, otherwise the thread that caused the creation of the object cannot resume its execution.

- The use of an early return should be avoided when the sequential execution of actions is important. Because the server cannot determine the requirements of the -future- clients, externally visible side-effects should be avoided after an early return.
- Allowing intra-object concurrency in general increases the amount of concurrency in the system. This may be attractive if an object is composed of multiple active parts; especially the types of objects that function as aggregation or subsystem objects are likely candidates to allow intra-object concurrency without this causing inconsistencies. Note that the parts of an object that are composed on the interface will accept messages concurrently.
- The danger of inconsistencies occurring when allowing multiple threads within an object is limited in the composition filters object model. Because all the nested objects (internals, instance variables, and temporary objects alike) are first-class objects that manage their own local synchronisation, inconsistencies are not likely to occur. The only situation that may lead to data inconsistencies is when a certain 'transaction' (i.e. a method body) performs both read and write actions on nested objects that can be simultaneously modified by other threads. The other type of inconsistencies deals with the synchronisation and sequencing of activities. This must be performed on the object boundary, which may require making calls to `self` or `server` within method bodies.
- In general, we do not promote the creation of concurrency for performance reasons: additional concurrency only increases performance on specific architectures[2], and we prefer not to make assumptions about this. In addition, a general application already incorporates sufficient concurrency to allow the application to be distributed over multiple processors. Only when performance limits are hit, it may be decided during a design iteration that an increased amount of concurrency is required for performance reasons. On a -massively- parallel architecture this may well increase the performance of the system. See [Rein 94] for a more extensive discussion on parallel architectures.

The example:

In the philosophers problem, intra-object concurrency is not very important: a philosopher can only do one thing at a time: think, collect chopsticks or eat. Collecting chopsticks, however, can be done in parallel at the left and at the right neighbour.

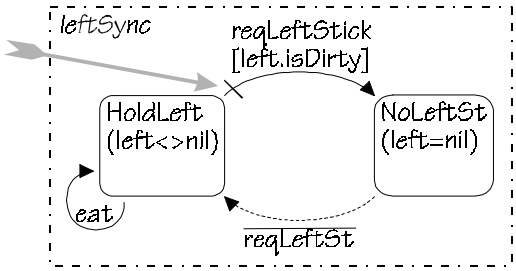


Figure 4.4.2 Increasing the amount of concurrency through an early return.

[2] On the contrary, as concurrency incorporates some overhead (e.g. task switching), on a single processor architecture increasing concurrency may even reduce performance.

To achieve this, a method that sends the reqXStick message to a neighbour philosopher must be defined such that it performs an early return before sending the message. In the graphical notation this is defined by the dash in the reqXStick transition (as indicated by the arrow), as illustrated by figure 4.4.2.

Another option for increasing the amount of concurrency in the system is by allowing intra-object concurrency in the philosopher objects, and restricting this so that only reqLeftStick and reqRightStick methods can be executed in parallel (but for each method only one execution at a time). This is shown in the ensuing figure. Four states are distinguished: the Free state means that no threads are active within the object: in this case all messages are allowed (except for the constraints that are defined by the other three SCDs). In the Busy state a message other than reqLeftStick or reqRightStick, or both are active within the object: no other messages are allowed until the message returns. The object is in state ReqLbusy (ReqRbusy) when the reqLeftStick (reqRightStick) method is executed. In this case only the reqRightStick (reqLeftStick) message is accepted (only when this happens we have concurrent activities within the object):

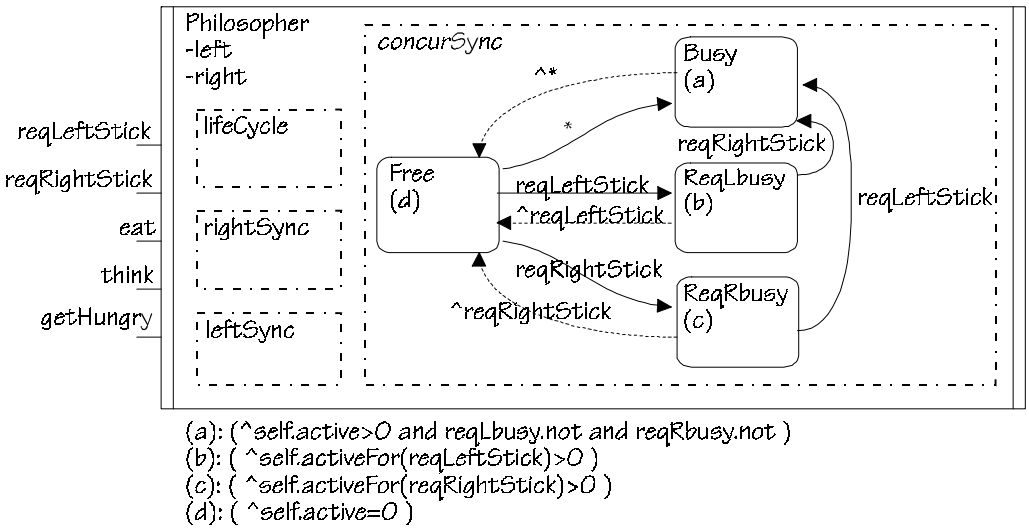


Figure 4.4.3 An example of intra-object concurrency for the Philosopher class.

The use of the caret '^' in the virtual transitions is an -informal- designation of message termination or reply; '^*m*' means the termination of message *m*. In the rest of this chapter we will not further consider intra-object concurrency in the philosopher example.

4.4.1 Step VII. Translating FODs to Composition Filters Specifications

Step VII: Translating FODs to Composition Filters Specifications.
Input: Full object diagram.
Output: A composition-filters template specification of the subsystem.
Side-effects: -
Description: The full object diagram is to be translated into an implementation in the composition filters model. This gives a template for the implementation of the complete application.

Description: The translation from FODs to a composition-filters specification is detailed in section 4.5. As is discussed there, this step can be performed fully automated, provided that the FOD specification is complete. The translation is rather straight-forward, and can also be performed by hand. The result of this step is a template of class descriptions for all the objects in the subsystem. The most significant gaps in these templates are the implementations of method bodies. The translation of the dining philosophers example is also demonstrated in section 4.5.

The example:

We show here a Sina template, which is the output of the translation process. The translation of the example is explained in section 4.5.

```
class Philosopher interface
    conditions
        Eating; Thinking; Hungry;
        HoldLeft; NoLeftSt;
        HoldRight; NoRightSt;
        HoldLeft_reqLeftStick; HoldRight_reqRightStick;
    methods
        reqLeftStick( ) returns ... ;
        reqRightStick( ) returns ... ;
        eat( ) returns ... ;
        think( ) returns ... ;
        getHungry( ) returns ... ;
        getLPhil( ) returns ... ;
        getRPhil( ) returns ... ;
        putLPhil( ) returns ... ;
        putRPhil( ) returns ... ;
    inputfilters
        lifecycle : Wait = { Eating=>think,
                             Thinking=>{reqRightStick, reqLeftStick, getHungry},
                             Hungry=>{reqRightStick, reqLeftStick, eat},
                             True~>{think, getHungry, eat, reqRightStick, reqLeftStick} };
        leftSync : Wait = { HoldLeft=>eat, HoldLeft_reqLeftStick=>reqLeftStick,
                             True~>{eat, reqLeftStick}  };
        rightSync : Wait = { HoldRight=>eat, HoldRight_reqRightStick=>reqRightStick
                              True~>{eat, reqRightStick}  };
        interface : Dispatch = { inner.* };
end // class Philosopher interface

class Philosopher implementation
    instvars
        left : Any;
        right : Any;
        leftPhil : Philref;
        rightPhil : PhilRef;
        Eating : Boolean;
        Thinking : Boolean;
        Hungry : Boolean;
    conditions
        Eating   begin  return Eating  end;
        Thinking   begin  return Thinking  end;
        Hungry   begin  return Hungry  end;
```

```
        HoldLeft  begin  return (left<>nil)  end;
        NoLeftSt  begin  return (left=nil)  end;
        HoldRight  begin  return (right<>nil)  end;
        NoRightSt  begin  return (right=nil)  end;
        HoldLeft_reqLeftStick  begin  return HoldLeft cand (left.isDirty) end ;
        HoldRight_reqRightStick  begin  return HoldRight cand (right.isDirty) end ;
    methods
        // we omit the method templates
end // class Philosopher implementation
```

4.4.2 Step VIII. Implementation

Step VIII: Implementation.
Input: A composition-filters template specification of the subsystem.
Output: A full composition-filters specification (e.g. a Sina program).
Side-effects: identification of new parts & new methods.
Description: The template specification that is delivered by step VII must be completed. This consists mainly of coding the method bodies.

Description: In this phase the templates that have been defined and generated are completed. This consists mainly of the coding of the method bodies. The implementations of methods are based on the specifications and documentation that has been collected in steps II (the interface of objects), IV (object interactions) and V.4 (event specification) mainly. During these implementation activities it may well appear that new parts (instance variables, as we are dealing with implementation issues) or new (local) methods are required.

The example:
We do not show the full implementation of the Philosopher and DynPhilosopher classes, as we are primarily interested in the synchronisation aspects, which appeared in the template that was shown in the previous step.

4.4.3 The Software Development Process.

A software development method consists not only of a description of the individual method steps. These steps have to be performed in a certain order, where some steps will be visited several times. In the beginning of this section, in figure 4.4.1, we outlined the various method steps. In this summary, as well in the discussion in the remainder of the section we adhered to one particular ordering. However, this sequence does in general not reflect the order in which a developer will go through the method steps. In this subsection the software development life-cycle (the primary ordering of method steps) and iterative development (redoing parts of the development cycle) are discussed.

The Software Development Life-cycle
The often cited distinction between analysis and design is that analysis describes 'what', and the design phase describes 'how'. There are two problems in constructing a method that follows this approach: firstly, a method cannot prescribe how to obtain the solutions to solve the 'how' problem: this is a creative process that depends on a lot of design

requirements. Secondly, once a solution to the 'how' is invented, the model that was built of the application in the analysis phase is to be extended. What is actually required here is an *analysis* of the invented solutions.

Thus the analysis phase is intended for modelling the application based on the real-world and requirement specifications, whereas in the design phase the (design-) solutions are analysed and modelled. This means that during analysis and design basically the same activities are performed, but with a different focus. The design phase is more extended, as it includes a reconsideration step where the software engineering properties such as reusability, reliability and maintainability of the constructed model are considered. The model may also be changed to improve these properties. This approach to analysis and design is outlined by the following figure:

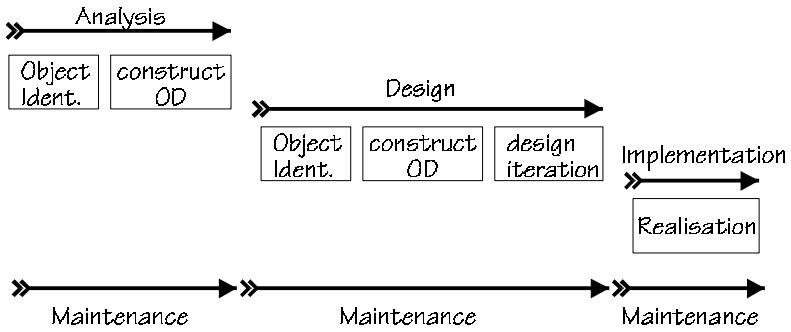


Figure 4.4.4 The software development life cycle.

Based on this figure we divide the software development life-cycle into the following four[3] stages:

Analysis: Based on the requirements specifications a real-world model of the application is constructed. This stage consists of three phases: the definition of a subsystem[4], object identification and construction of object diagrams. Concurrency and synchronisation issues may well appear in this phase if they are required for appropriate modelling of the real-world system.

Design: The developer must construct solutions to the 'how' problems that are left open by the analysis phase. These problems may include searching or sorting problems, numerical algorithms, inventing efficient storage structures, increasing system performance e.g. by caches or hashing tables, and synchronisation problems. The solutions to the problems will require the addition of new objects, new structural relations, extended object interfaces, etcetera. Thus the design stage comprises object identification and the construction (or modification) of object diagrams. Finally, the resulting model is reconsidered with typical design issues in mind: extensibility, reusability, maintainability, reliability, minimisation of inter-object dependencies etcetera. The more complex synchronisation problems are

---

3 We omitted the *testing* stage from this list, mainly because it is not very different from testing conventional systems.

4 This is not really different from defining new objects, and will usually be available as the result of a previous activity, except for the root subsystem.

typically solved during the design stage ('how' to realise the synchronisation of some activities).

Implementation: This is the realisation of the design. It incorporates code generation, resulting in object templates, and a programming step, which fills in the gaps that are left in the templates. The implementation should be straightforward: any reconsideration of the application model is considered to be design practice.

Maintenance: We consider maintenance to be a form of iteration over the developed application model. It may start at any particular phase and step of the method, and must necessarily continue up to the final implementation step in order to maintain a fully defined and consistent system. Thus, maintenance may range from fixing a typing mistake up to extending the requirements specifications, leading to the identification of new objects.

Iterative development cycle

We already saw that certain method steps will be performed more than once during the development cycle. The most important reason for making iteration explicit is that the results of a method step will hardly ever be permanent: as the system extends, and other method steps are performed, changes are very likely to occur. The required process flow is shown in the following figure:

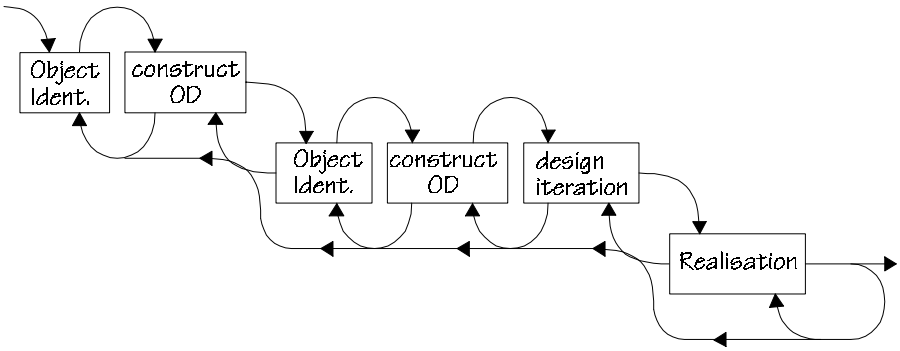


Figure 4.4.5 Iteration: the ordering of method steps may follow the arrows.

The figure shows that at any point during the development one may go back to one of the previous steps. However, when making changes to the application model in a method step, one should work out the effects the changes may cause in subsequent steps.

Apart from re-doing a certain step for some given subsystem, each step is to be performed repeatedly for all the subsystems that are nested within the current subsystem. Thus, we distinguish two axes of iteration: a horizontal axis, that repeats the same steps for the same subsystem, and a vertical axis, which repeats the software development life cycle for nested subsystems (this process can be recursive).

This does complicate the ordering of activities for the software developer: after each step in the process, there are several options to select the next activity to be done:

- ❏ Proceed with the next method step, on the same subsystem.
- ❏ Start the development cycle for one of the subsystems.
- ❏ Go back to the encapsulating subsystem, and select an activity.

- Go back to a previous method step (i.e. iterate).

We do not address the issues involved in multi-developer projects here.

An iterative approach is suitable for prototyping systems: this means that certain aspects of the system (say, a particular subsystem) can be worked out independently up to the realisation phase. Meanwhile, iterations for that particular subsystem can be performed. Such an approach to the development of a large system has important benefits: it allows to test, demonstrate and verify certain -critical- parts of the system, before committing on it, and basing other parts of the system on the committed version.

4.5 Generating Composition Filters Specifications from FODs

In previous sections we have emphasised the importance of a straightforward translation from the (analysis and design) diagrams to the implementation model. In the absence of a clear mapping between these two, the effort put into the construction is partially lost, as the developer has to re-invent a solution, this time expressed in a notation that suits the implementation model. We have made an effort to provide the developer with a notation that is both suitable for the analysis and design phases, and has a well-defined mapping to the composition-filters model, which is our preferred implementation model.

The most important exception to this is constituted by the implementation of method bodies: for reasons that we outlined before we do not generate method implementations. Therefore we cannot translate interactions relations, causal transitions and state changes.

In this section we describe how the translation can be made from the full object diagrams to the composition filters model. First we give an informal description, based on example translations. Then we give a summary where for each component in the object diagrams an equivalent in the composition-filters computation model is described.

4.5.1 Informal Description of the Translation

In this subsection we will discuss how all important components of full object diagrams can be translated to a composition-filters specification. We will use Sina code for the representation of the composition-filters specification. To illustrate the translation, we use the dining philosophers problem that was introduced and worked out in section 4.4. The result of the translation will be a Sina code template, with its full synchronisation specification defined, and as the most significant missing part the implementation of method bodies.

We start with a very simple example: the Chopstick object. Its object diagram is shown in the following figure. This diagram defines only two things: each chopstick object has a nested part clean, and three methods, use, clean and isDirty:

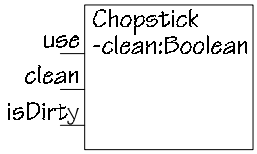


Figure 4.5.1 A simple class with only methods and nested objects.

The translation of this diagram is given here, interleaved with comments on specific issues:

```
class Chopstick interface
    methods
        use( ) returns ... ;
        clean( ) returns ... ;
        isDirty( ) returns ... ;
```

This is a part of the translation of the methods on the interface of the object. Only for the input interface this is done. Note that we -explicitly- ignored the full definition of the interface methods. Otherwise information such as the types of arguments and return values could be used here to generate more complete method specifications.

inputfilters
interface : Dispatch = { inner.* };

All the methods that are defined locally are made available on the interface of the object.

end // class Chopstick interface

class Chopstick **implementation**
instvars
clean : Boolean;

The nested object (attribute) clean is mapped to an instance variable of the specified type: because it does not appear on the interface of the object, it can be fully encapsulated in the implementation part.

methods
use( ) **returns** ... **begin** ... **end**;
clean( ) **returns** ... **begin** ... **end**;
isDirty( ) **returns** ... **begin** ... **end**;

These are the templates for the methods defined by this class.

end // class Chopstick implementation

This is the template for class ChopStick; it provides the structure for the definition of the class, without much detail. Only for synchronisation specifications the implementation will be generated in detail.

The next example is concerned with the translation of the Philosopher object, which features an extensive synchronisation specification:

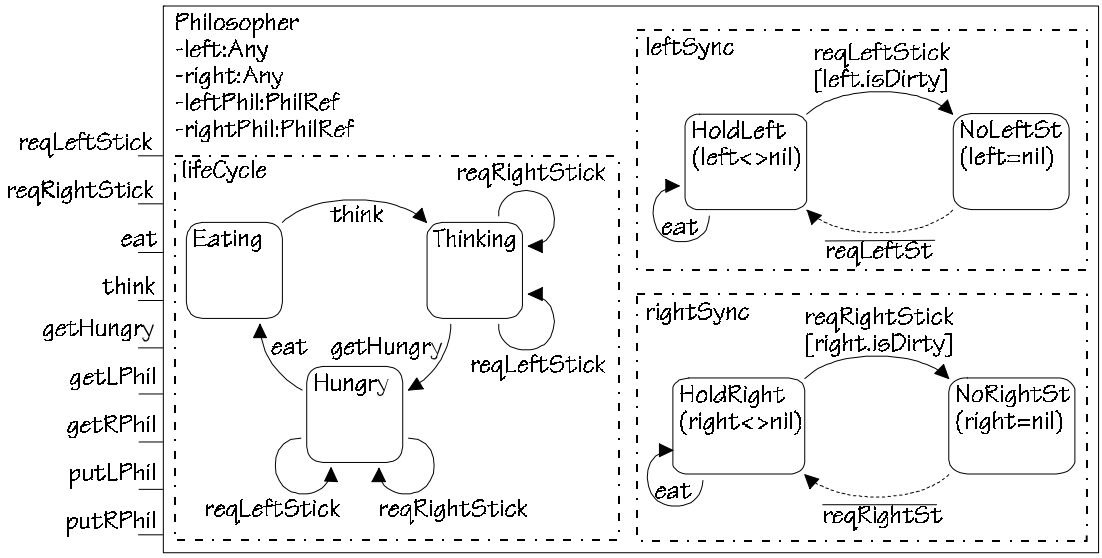


Figure 4.5.2 The FOD of the Philosopher class.

The translation, interleaved with the explanations, is as follows:

class Philosopher **interface**
conditions
Eating; Thinking; Hungry;
HoldLeft; NoLeftSt;
HoldRight; NoRightSt;

For each of the states in all the SCDs, a condition with the same name is declared. The implementation of the conditions is in the implementation part. The purpose of the following two

condition declarations will be explained at the filters where they appear. For now we suffice by saying that these combine a state with a condition on a transition:

```
HoldLeft_reqLeftStick;          // more condition declarations
HoldRight_reqRightStick;
```

The declaration of the methods on the interface of the object:

```
methods
    reqLeftStick( ) returns ... ;
    reqRightStick( ) returns ... ;
    eat( ) returns ... ;
    think( ) returns ... ;
    getHungry( ) returns ... ;
    getLPhil( ) returns ... ;
    getRPhil( ) returns ... ;
    putLPhil( ) returns ... ;
    putRPhil( ) returns ... ;
inputfilters
    lifecycle : Wait = { Eating=>think,
                         Thinking=>{reqRightStick, reqLeftStick, getHungry},
                         Hungry=>{reqRightStick, reqLeftStick, eat},
                         True~>{think, getHungry, eat, reqRightStick, reqLeftStick} };
```

This wait filter specification is the translation of the lifecycle SCD in the diagram above: the first property of this SCD that should be noted is that it does not contain any wild cards. This means that all messages that are not explicitly specified in the SCD should pass this filter without constraints. This is achieved by the last line of the filter specification: "True~>{..}". In the three lines before that, for each of the states Eating, Thinking and Hungry all the messages that are allowed in that particular state are specified. For instance, when the object is in state Eating, only the think message is allowed, all other messages are blocked on the interface of the object.

```
    leftSync : Wait = { HoldLeft=>eat, HoldLeft_reqLeftStick=>reqLeftStick,
                        True~>{eat, reqLeftStick}  };
```

This wait filter implements the leftSync SCD. The filter largely follows the structure that we saw for the lifecycle filter; at the end of the filter we allow all messages that the SCD is not concerned with to pass without constraints. For all states in the SCD we associate the corresponding condition with the messages that are accepted in that state. An exception is made for those messages that have an additional constraint (as specified between the square brackets). In this case the messages that the leftSync SCD deals with, eat and reqLeftStick, are only accepted when in the HoldLeft state. But reqLeftStick has an additional constraint, "left.isDirty". Thus the only message associated with HoldLeft is the eat message. With the reqLeftStick message a specialised condition, HoldLeft_reqLeftStick, is associated, that is satisfied only when both the HoldLeft condition and the "left.isDirty" constraint are satisfied.

The outgoing message $\overline{\text{reqLeftStick}}$ is not synchronised because the arc is drawn as a virtual transition. Otherwise, in the outputfilters of the object, this synchronisation should be specified (analogous to the input filters). The translation of the rightSync SCD is analogous to the leftSync SCD:

```
    rightSync : Wait = { HoldRight=>eat, HoldRight_reqRightStick=>reqRightStick
                         True~>{eat, reqRightStick}  };
    interface : Dispatch = { inner.* };
```

The only messages that are supported are those that are defined in the implementation part.

```
end // class Philosopher interface
```

```
class Philosopher implementation
    instvars
        left : Any;
        right : Any;
        leftPhil : Philref;
        rightPhil : PhilRef;
```

All the nested objects of Philosopher are encapsulated as instance variables. Three additional instance variables are defined to model the states the object can be in:

```
        Eating : Boolean;
        Thinking : Boolean;
        Hungry : Boolean;
conditions
    Eating  begin return Eating end;
    Thinking  begin return Thinking end;
    Hungry  begin return Hungry end;
```

The implementation of the previous three conditions can be generated automatically: when no Boolean expression is specified for a state in a SCD, the assumption is made that there is a one-to-one mapping between the state and a Boolean variable. It is important to note, however, that all the transitions in the SCD to and from that state are responsible for updating these Boolean variables. In this case these are the eat, think and getHungry methods.

```
        HoldLeft  begin return (left<>nil)  end;
        NoLeftSt  begin return (left=nil)  end;
        HoldRight  begin return (right<>nil)  end;
        NoRightSt  begin return (right=nil)  end;
```

The implementation of these methods is directly derived from the state specifications in the SCD. We show the NoLeftSt and NoRightSt conditions here, although these are not used in the filters of this object. However, it is still possible that a -future- subclass refers to one of these conditions. Therefore, they must be implemented.

```
        HoldLeft_reqLeftStick  begin  return HoldLeft cand (left.isDirty) end ;
        HoldRight_reqRightStick  begin  return HoldRight cand (right.isDirty) end ;
```

The implementation of these two conditions is derived from the start state of the transition, ANDed with the constraints that are specified for the reqLeftStick and reqRightStick transitions, respectively.

```
    methods
        reqLeftStick( ) returns ...
            begin ... end;
        reqRightStick( ) returns ... ;
            begin ... end;
        eat( ) returns ... ;
            begin ... ; Hungry :=false; Eating := true; ... end;
        think( ) returns ... ;
            begin ... ; Eating :=false; Thinking := true; ... end;
        getHungry( ) returns ... ;
            begin ... ; Thinking :=false; Hungry := true; ... end;
```

In the bodies of these methods the code for maintaining the variables that model states is inserted. Note that it is the responsibility of the developer/programmer to embed this code in the functional method code[1]. The other methods are to be implemented completely yet:

---

[1] A more robust approach would be to generate an ACT object that updates the state variables whenever appropriate: this can be achieved by reifying and delegating received messages just

```
        getLPhil( ) returns ... ;
            begin ... end;
        getRPhil( ) returns ... ;
            begin ... end;
        putLPhil( ) returns ... ;
            begin ... end;
        putRPhil( ) returns ... ;
            begin ... end;
end // class Philosopher implementation
```

As the final example, we show the translation of the DynPhilosopher, which incorporates inheritance. First the diagram is shown:

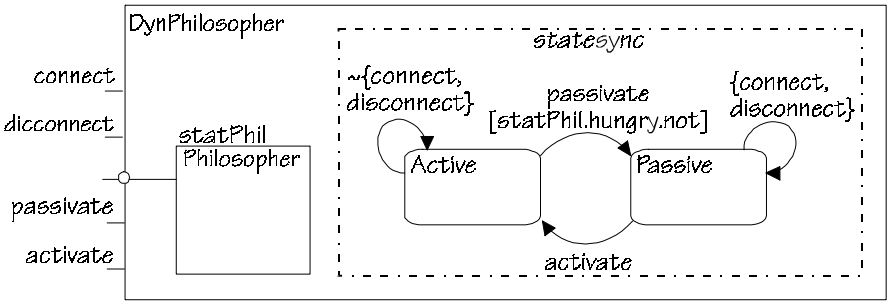


Figure 4.5.3 The FOD of the DynPhilosopher class.

Except for inheritance, this class features another interesting issue: the use of wild cards in the synchronisation specification affects the manner in which the Wait filters are constructed:

```
class DynPhilosopher interface
    internals
        statPhil : Philosopher;
```

The statPhil object appears as an internal rather than an instance variable, as it is made available on the interface of DynPhilosopher by a reuse relation. This reuse relation is translated into an appropriate definition of the interface (Dispatch) filter below.

```
    conditions
        Active; Passive;
        Active_passivate;
    methods
        connect( ) returns ...;
        disconnect( ) returns ...;
        passivate( ) returns ...;
        activate( ) returns ...;
    inputfilters
        stateSync : Wait = { Active ~>{ connect, disconnect, passivate},
                             Active_passivate=>passivate,
                             Passive=>{connect, disconnect, activate} };
```

This filter implements the synchronisation defined by the stateSync SCD. Because the wild card (exclusion) operator appears in the SCD, the synchronisation applies to all the messages that

---

before dispatching them to this ACT, which modifies the state variables corresponding to the particular messages. Subsequently, the messages are fired and really dispatched to the appropriate methods. This is similar to the implementation of the HistoryBuffer in 3.4.3.

arrive at the object, and no 'bypass' is needed. Thus in the filter the associations state-event can be straightly mapped to condition-message associations. This is done in the first (which features wild card exclusion) and third filter element.

In the first filter element the passivate message is excluded as well, since this message is only satisfied when the Active_passivate condition is satisfied. This is expressed by the second filter element.

```
interface : Dispatch = { inner.*, statPhil.* };
```

The reuse relation is implemented by this Dispatch filter: except for the local methods ("inner.*"), all (as this is the default) methods of the statPhil object are made available on the interface.

```
end // class DynPhilosopher interface

class DynPhilosopher implementation
    instvars
        Active : Boolean;
        Passive : Boolean;
```

The implementation part of DynPhilosopher mainly deals with the management of the states of the object, as expressed by the Boolean instance variables Active and Passive.

```
    conditions
        Active  begin  return Active  end ;
        Passive  begin  return Passive  end ;
        Active_passivate begin  return Active cand (statPhil.Hungry.not)  end ;
    methods
        connect( ) returns ...
            begin ... end;
        disconnect( ) returns ...
            begin ... end;
        passivate( ) returns ...
            begin ... ; Active := false; Passive := true; ...  end;
        activate( ) returns ...
            begin ... ; Passive := false; Activate := true; ...  end;
```

The code for updating the variables that represent the states Passive and Active is inserted into the bodies of the activate and passivate methods.

```
end // class DynPhilosopher implementation
```

This subsection demonstrated the translation of some complete object diagrams to a composition filters implementation. This showed the translation of the most relevant constructs. In the following subsection for all the components in object diagrams their translation is discussed.

4.5.2 The Translation of Object Diagram Components

We will now describe the translation of full object diagrams by discussing for each component of the diagrams how it is translated to the composition filters model. We take the abstract syntax of full object diagrams, and discuss the components that are specified by it one by one (sometimes grouping and re-ordering small sets of components for convenience):

FullObjectDiagram $\stackrel{def}{=}$ *objectSet:LabeledEntities; relSet:Relations; interaction:MessConns.*

In an object diagram a set of objects is defined, among with a number of relations that connect them. The structural relations must all be converted into compositional relations (nesting and reuse relations). The interaction relations are not translated, as these are implemented in the bodies of methods.

LabeledEntities $\stackrel{def}{=}$ *LabeledEntity*.*

LabeledEntity $\stackrel{def}{=}$ *label:Identifier ; entity:Entity.*

An object diagram describes a subsystem. As a subsystem is an object itself, the objects in the subsystem are a *part-of* the subsystem object. Each object has an identifier *label*, and a class that is defined by *Entity*.

Entity $\stackrel{def}{=}$ *ObjectCollection* | *Object* .

ObjectCollection $\stackrel{def}{=}$ *objectDef:Object* .

An *Entity* can be either a collection of objects, or a single object: when it is a collection, a collection of *objectDef* objects is defined within the subsystem. Then (in both cases) for the definition of the object a class (template) is generated.

Object $\stackrel{def}{=}$ *className:Identifier; attribs:Attributes; interface:Interface; parts:Parts; reuse:ReuseRels; parallel:*𝔹*; scd:SCDiagram.*

It must first be checked whether the class *className* already exists: when this is the case, the existing class definition must be extended, otherwise a new class definition is generated. The *parallel* component determines whether the mutual exclusion filter is to be inserted or not. The other components are discussed below:

Attributes $\stackrel{def}{=}$ *Attribute*.*

Attribute $\stackrel{def}{=}$ *label:Identifier; type:Identifier* .

Attributes correspond to instance variables, as they are not connected to the interface of the object (no reuse relation with attributes). The *label* and *type* are used for proper instance variable declaration.

Interface $\stackrel{def}{=}$ *input:MessSet; output:MessSet* .

For each message that is defined by the input interface, a method declaration and a method implementation template are generated. As the FOD does not define arguments and return types, a full declaration cannot be generated. The messages of the output interface have no direct mapping to the composition-filters model (outgoing messages need not be declared); they are the result of message invocations in method bodies.

Parts $\stackrel{def}{=}$ *LabeledEntity*.*

For each of the part objects, when they are connected to the interface of the object with a reuse relation, they are translated into internal declarations, otherwise to instance variable declaration. The part object can be a newly defined object itself, which requires the recursive application of the translation algorithm.

ReuseRels $\stackrel{def}{=}$ *ReuseRel** .

ReuseRel $\stackrel{def}{=}$ *mess:MessSet ; obj:ObjectRef.*

All reuse relations are realised by a single Dispatch filter: the definition of this filter is as follows:

```
interface : Dispatch = { inner.*, .... };
```

The first element in the filter definition ensures that the locally defined methods are available on the interface of the object as well (possibly overriding reused methods). This is followed by the reuse relations: *obj* is the target of the filter elements, for each of the messages in set *mess*. All target objects must be either internals (i.e. part objects), or are declared as externals.

Relations $\stackrel{def}{=}$ *Relation*.*
Relation $\stackrel{def}{=}$ *Inheritance | Delegation | Aggregation .*
Inheritance $\stackrel{def}{=}$ *superclass:EntityRef; subclass:EntityRef .*
Delegation $\stackrel{def}{=}$ *delegatee:EntityRef; delegator:EntityRef .*
Aggregation $\stackrel{def}{=}$ *part:EntityRef; whole:EntityRef .*

These structural relations are transformed into composition relations as defined in subsection 4.2.3. Inheritance is transformed into a combined part-of and reuse relation, delegation is mapped to a reuse relation only, and aggregation of an object is equivalent to defining the object as a part.

MessConns $\stackrel{def}{=}$ *MessConn*.*
MessConn $\stackrel{def}{=}$ *direction:Direction ; from:EntityRef; to:EntityRef ; mess:MessSet.*
Direction $\stackrel{def}{=}$ *Right | Left | BiDir.*

Message connections are not translated (as they are implemented as message invocations that are woven into the bodies of methods).

SCDiagram $\stackrel{def}{=}$ *SCDunit*.*
SCDunit $\stackrel{def}{=}$ *label:Identifier; states:States; transitions:Transitions.*

A single object definition may consist of a number of SCD(-unit)s. For each SCD, at least one Wait filter is generated. When no wild cards appear in any of the transitions in the diagram, a filter element is created that enables all messages (m1, ... mx) that are not syntactically available in the SCD:

```
scdLabel : Wait = { True~>{m1, ...mx}, ... }
```

The other elements of the filter are the translation of the transitions in the SCD:

Transitions $\stackrel{def}{=}$ *Transition* .*
Transition $\stackrel{def}{=}$ *from:Identifier; to:Identifier; spec:TransSpec ; cond:Expression*$_{CFM}$ *;*
type:TransitionType .

The translation of the transition depends on the type of transition that is dealt with; therefore we describe the translation separately for each individual transition type below. Common to these translations is that the *to* component never appears in the filter specification, and that the presence of a constraint *cond* results in the creation of a new condition. The condition is labelled as the concatenation of the *from* state and the message label (another label is constructed in case of duplicate names or when the transition represents multiple messages). The definition of the condition is equivalent to "*from* AND *cond*"; only when the object is in state *from* and the expression *cond* is satisfied, the associated messages *spec* are accepted.

TransSpec $\stackrel{def}{=}$ *excl:ExclSpec; set:MessSet.*
ExclSpec $\stackrel{def}{=}$ *Exclude | Permit .*

This is the specification of the message set that the transition covers: the *set* can consist of one (e.g. "m") or more (e.g. "{m1, m2, m3}") messages, the *excl* component is mapped to the "~>" (*Exclude*) or "=>" (*Permit*) operator.

TransitionType $\stackrel{def}{=}$ *PlainTrans* | *MutexTrans* | *VirtTrans.*

PlainTrans $\stackrel{def}{=}$ *action:Expression*$_{CFM}$.

PlainTrans describes a normal transition; this consists of the condition defining the *from* state, followed by the translation of the *spec* message specification. Examples of the filter elements that are generated as the result of the translation are "State1=>m" or "State2~>{m1, m2, m3}". The action annotations are in principal not translated, although the expressions can be put within the bodies of the methods that are to be executed.

MutexTrans $\stackrel{def}{=}$ *action:Expression*$_{CFM}$.

A mutual exclusive transition is a transition which defines two additional synchronisation specifications: firstly, the transition can only be made once there are no other activities within the object. Secondly, all other transitions in the object are blocked until this transition is finished. These two additional synchronisation constraints are defined in a separate filter. The translation of the transition itself is similar to the translation of normal transitions. The additional filter for, say, message m is defined as:

```
mutex_m : Wait ={ Free=>m, No_m~>m };
```

Where the conditions Free and No_m are defined as:

```
Free begin return ^self.active=o end;
No_m begin return ^self.activeFor(m)=0 end;
```

This filter ensures that message m is only accepted when the number of active threads in the object is zero, and all messages except m are accepted when there is no thread in the object active executing m.

VirtTrans $\stackrel{def}{=}$.

As this is a virtual transition, no synchronisation specification is generated for it; virtual transitions are only provided to make the SCDs more readable and complete.

States $\stackrel{def}{=}$ *State** .

State $\stackrel{def}{=}$ *id:Identifier; spec:Expression*$_{CFM}$.

The state specifications are used to generate conditions that model the states the object can be in: *id* is used as the condition identifier, and *spec* as the implementation of the condition.

This informal discussion of the translation can presented in a more precise form as a translation algorithm. The algorithm can be seen as the translational semantics from the FOD specification to the composition-filters computation model (CFM). We omit such a presentation here as it requires a significant amount of spaces but does not introduce new perspectives.

4.6 Conclusion

In this section we will evaluate the method that was presented in this chapter. First, we will compare the method and its notation to the requirements and guidelines that we stated in section 4.1. Secondly, we will compare our approach to a few publications that we consider as related work. Then we will discuss some directions for further research, and finish with a discussion of the contributions that are made in this chapter.

4.6.1 Comparison with Requirements.

We will now one by one discuss the requirements for methods as proposed in section 4.1:

1. *Support for modelling and solving concurrency issues:* This support is provided through a number of method steps that offer a view on the application that is suitable for considering concurrency issues. In particular the presentation of the application as a configuration of interacting objects, and making the nesting of objects for reuse explicit, is intended to provide an informative diagram for considering concurrency and synchronisation. The SCD notation offers an intuitive, yet precise, notation for modelling synchronisation constraints. In addition the steps of the method address a wide range of hints and clues that help in finding and resolving synchronisation issues.
2. *Integration with OO principles and the CF model:* The model that we propose strictly adheres to the object-oriented model; objects are the unit of synchronisation, and synchronisation is performed on messages. Most importantly, the synchronisation specifications are well integrated with inheritance and delegation relations, both between objects, and between the synchronisation specifications themselves. Encapsulation and polymorphism are fully supported. The only data modelling aspect in the method that is specific to the composition filters model is the view on reuse and inheritance, the method supports the transformation between this model and the conventional object-oriented view. The translation of synchronisation specifications to the composition filters model is addressed elaborately.
3. *Support open-endedness and extension:* The properties of open-endedness and extensibility are obtained by constructing a model of objects that conforms to the important composition-filters object model properties. Refer to the conclusions of chapter 3 for a discussion why this model provides improved open-endedness and extensibility properties. The semantics of SCDs, which allow them to be specified in either the synchronise-set-only or the synchronise-for-the-future approach (see subsection 4.4.8), also contribute to the open-endedness.
4. *Support Reuse:* Reuse is supported by two aspects; firstly it is made possible because of the open-endedness and extensibility properties of the underlying model, as was just discussed. Secondly, the method promotes reuse by addressing the issue explicitly at various stages during the development. Reuse is supported along four axes[1]; reuse of

[1] In fact, the composition-filters model supports reuse along the following 4 axes: data reuse (object state), behaviour reuse (message interface), condition reuse (abstract object states) and filter reuse. We specialise this to focus on synchronisation.

behaviour of objects (inheritance), reuse of data of objects (through message connections or delegation), reuse of states and reuse of complete SCDs.

In order to support extensibility of synchronisation specifications, our style of specification can be characterised as *programming-the-difference.* An essential property in obtaining reuse is the ability to *compose* specific aspects from pre-existing objects. In our case, this reveals itself through the ability of fine-grained synchronisation specification reuse, such as reuse of state specifications.

5. *Reduce the implementation gap:* By fully specifying the translation from the graphical design notation to a composition-filters specification, the implementation gap is largely bridged: the main gap that is remaining is the implementation of methods, plus some domain-specific aspects that are not considered here, such as real-time specifications, query facilities, atomic delegations, abstraction of object interaction, etcetera. The translation can be done by hand, or with automated tools.
6. *Support iterative development:* Iterative development is explicitly supported and promoted by the development process. Each step can be performed either as a creating step or as an adaptation of an existing component. An important property for developing CASE tools that support the method is the precise definition of notation and semantics for the various diagrams. This allows for the precise specification of the effects that changes in one diagram have on other diagrams that represent the same objects or subsystem.
7. *Intuitiveness:* An attempt has been made to achieve intuitiveness, but it is difficult to verify how successful this attempt has been. Examples are the introduction of visual representations of objects, relations between objects and, the most important contribution, synchronisation specifications. In addition, the steps and substeps presented by the method are intended to let the developer construct, step-by-step, a model of the application. The various aspects, such as object properties, structural and interaction relations, and the synchronisation specification are incrementally addressed. The steps are presented in an order such that each step supplements the context for successfully performing the successive method step.

In addition to the requirements for methods, in section 4.1. a number of requirements on notations were defined. We will not discuss each of these requirements, as they are guidelines rather than concrete and exact requirement specifications. As a result it is hard to make statements about success or failure to meet the requirements. We will confine ourselves to an informal discussion about the properties of the notation, with a few examples to illustrate these.

The elements in our notations are all simple geometrical shapes, the amount of different symbols is limited: for entities in the notations, such as objects, SCDs and states, different types of rectangles are used. Relations between entities, such as structural relations, interaction relations, reuse relations, and transitions between states, lines and arrows are used, with a few icons to distinguish structural relations. Entity nesting can always be expressed by drawing one entity within the boundaries of the other. We have taken care that all elements of the notation are suitable to be drawn by hand, the use of tools for drawing the diagrams should not give any problems.

In addition, for each type of diagram we have provided an exact specification of the components, and each component has unambiguous semantics. The latter is illustrated by defining the translation to the composition filters model (this could even be changed into translational semantics for the formal representation of the notation). The precise semantics also allow to define a mapping between the different diagrams; components in different diagrams are either exactly the same, they are fully orthogonal (such as in structural relation diagrams and object interaction diagrams), or they are related to each other, in which case a mapping between the two representations exists (such as for inheritance, delegation and part-of relations).

4.6.2 Comparison with Related Work

No publications we know of have exactly the same goals as the method presented in this chapter, namely providing methodological and notational support for the invention, derivation and reuse of synchronisation specifications in object-oriented analysis and design. We will suffice in a comparison with the related work on methods that was discussed in section 4.1.

The Eiffel// method, although it addresses roughly the same area of research, does hardly pay attention to the analysis and specification of synchronisation specifications. The issue of data consistency and data sharing is discussed, but its influence on synchronisation specifications is not investigated. In addition, the properties of the underlying computation model make effective reuse of synchronisation specifications difficult (see chapter 3 for a more detailed discussion).

The Object Lifecycles method provides rather elaborate support and notations in the analysis phase, but the authors explicitly state that they are unable to outline a realisation scheme for concurrent environments. Most other object-oriented analysis and design methods suffer from the same problem: for instance for the OMT method [Rumbaugh 91], the translation from the dynamic model to an implementation model is described in [Rumbaugh 93], following roughly the same approach as the Object Lifecycle method. The issue of concurrency is neglected in [Rumbaugh 93].

For the development of the parallel object-oriented operating system CHOICES a notation based on control flow graphs was applied for describing the dynamic aspects of frameworks [Campbell 93]. No specific support for the specification of synchronisation constraints is provided.

4.6.3 Further Research

By no means the work presented in this chapter is finished, although the method is rather elaborate in its presentation of those aspects of an object-oriented method that focus on concurrency, particularly synchronisation aspects. However, to make the step to a complete method that can be applied in practice, more attention must be paid to the conventional object-oriented analysis and design. A number of specific topics need to be addressed more elaborately as well. For instance, the identification and specification of ACTs, an explicit notion of distribution aspects, and the combination of atomic delegations and synchronisation.

Another interesting issue is the integration of the method presented here within the so-called *hermeneutic* approach to software development. Hermeneutics provide a framework to explicitly model the various design decisions, hints and alternatives, and support these in a computer-aided software engineering environment. An important property of hermeneutics is that all development information is kept and used to actively support iteration and maintenance. More information on hermeneutics can be found in [Aksit 94c], [Koehorst 94] and [Algra 94].

A different topic for future research is the exploitation of the state composition diagrams: in this chapter these are used only to express synchronisation constraints. However, the diagrams actually are a visual representation of filter specifications. This suggests that the same notation, or a closely related one, can be used to visualise the specification for arbitrary filter types, such as dispatch and error filters. It is not clear at this point to what extent this is possible, and whether such a notation will be effective for these domains.

4.6.4 Contribution

The method that has been introduced in this chapter has the following contributions:

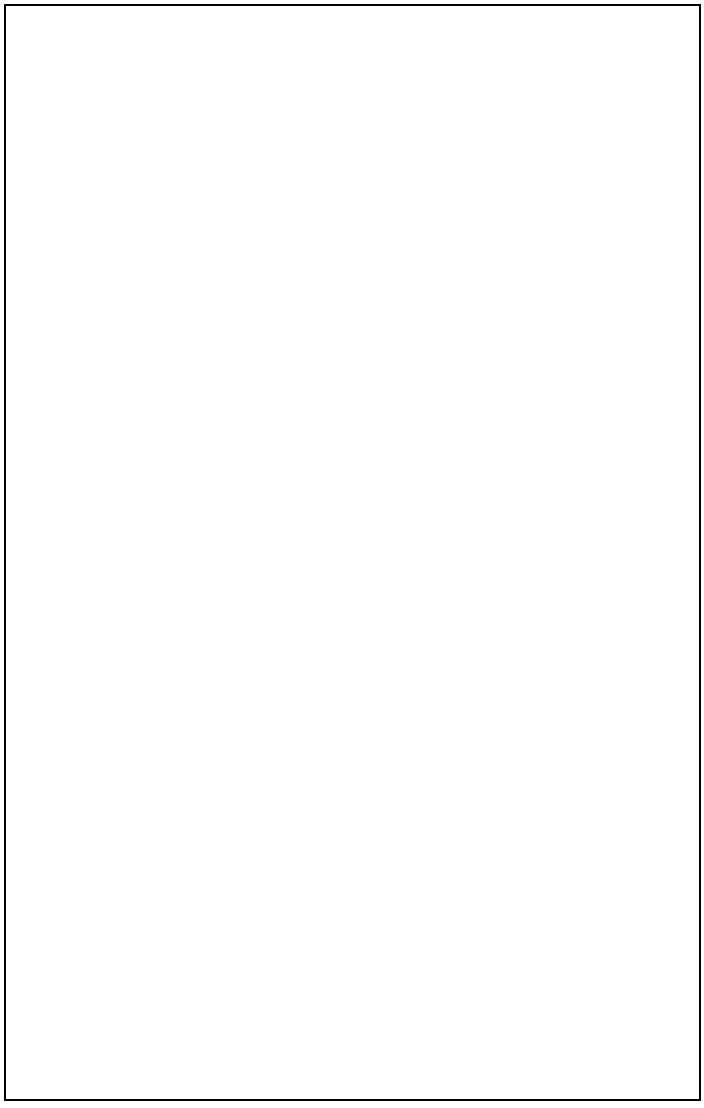
- ❏ A novel graphical notation for expressing synchronisation constraints, closely related to the well-known paradigm of state-transition diagrams, offers an intuitive presentation of the life cycle of an object and the effect of the object state on the synchronisation of messages.
- ❏ The graphical notation has well-defined semantics, which are expressible in terms of the composition filters computation model.
- ❏ The notation is composable, which allows synchronisation specifications to be split in a number of independent diagrams. These diagrams can be freely combined and reused.
- ❏ The implementation gap between the graphical notation and the implementation model is bridged with a translation algorithm. This provides important support for incremental development
- ❏ An object-oriented analysis and design method is introduced with extensive support for synchronisation aspects. This includes the analysis of synchronisation issues, the design of synchronisation strategies, the derivation of synchronisation constraints and the detection of consistency problems. An important aspect is that the method supports the invention of new, tailored synchronisation specifications, rather than being a discussion of mostly well-known, pre-defined synchronisation problems and their solutions.
- ❏ The method is integrated within the object-oriented paradigm, and it promotes and supports the construction of reusable and extensible objects.

One important issue that we would like to stress here is the following: An attempt has been made in this chapter to present a method that is both self-sufficient and complete, in the sense that the entire path from analysis up to maintenance is addressed. On the other hand, we emphasise that we are strongly focused on the issue of specifying synchronisation constraints for objects in a reusable and extensible manner. A wide range of other aspects of object-oriented software development needs to be addressed as well.

Most of the existing techniques and methods that do so can be easily integrated with the presented material, however. For example, design rules (e.g. [Johnson 88a] and [Lieberherr 89]) for improving the structure of the application classes can be applied

directly. The structure we brought into the notations allows for adding additional notations in an integrated manner, such as timing diagrams [Booch 90] or even associations [Rumbaugh 91]. Fully integrating such notations and techniques with all subsequent steps in the development may be difficult, though.

Chapter 5



Implementation Aspects

5.1 Implementation Issues

5.1.1 Introduction

The aim of this chapter is to argue that the proposed tools in the previous chapters, in particular the wait filter mechanism in chapter 3, can indeed be implemented effectively. Attention is paid in particular to the efficiency of implementations and our approach towards efficiency.

One of the important considerations in implementation is performance. Since our primary objective is to offer the software engineer expressive power that is practically applicable, we do not simply discard mechanisms when they are difficult to implement efficiently. This issue is considered elaborately in the next subsection.

5.1.2 Our Approach towards Performance

We define efficiency as the extent to which a program claims the available resources. The most important resources are usually memory and processor cycles. In many cases these two complement each other: by sacrificing memory, speed can be increased (e.g. through caching mechanisms) and memory can be conserved through -repeated- computations. It is common practice, though, to consider speed as the most important issue when making design decisions. In other words, processor cycles are considered to be the most scarce resource in a computer system. This is exemplified by the importance of caching mechanisms in current implementations (cf. the implementation of SELF [Chambers 92], [Hölzle 91]).

The language designer is frequently confronted with design decisions that require a trade-off between efficiency and expressive power. We use the term expressive power in this discussion to denote any mechanism or technique that relieves the burden of the programmer in some manner. This includes mechanisms like garbage collection, dynamic binding, reuse mechanisms, etcetera. When making these design decisions, often techniques with more expressive power are discarded because they compare poorly with respect to implementation efficiency. The usual motivation for this is that a system with very bad efficiency characteristics is very unlikely to have any practical impact.

Although we agree with the motivation for such decisions, we feel that sometimes a potential reduction in performance is inappropriately used as an argument to discard expressive power. In many cases, potentially inefficient techniques can well be adopted without seriously affecting all-over system performance. The following arguments support this claim:

- First of all, some forms of expressive power are very valuable, and are thus worth even a significant reduction in efficiency. An example of this is garbage collection: although this incurs significant overhead, it is considered an invaluable tool for object-oriented programming [Meyer 88].
- Second, related to the previous item, although most of the mechanisms in programming languages can be 'simulated' through application code as well, this will in many cases conflict with reusability and extensibility properties. For example, the mechanism of

delegation can be simulated by defining for each message that is delegated by an object a separate method, which sends a message to the delegated object, providing *self* (i.e. server in Sina) explicitly as an argument. Obviously, this can achieve the goals of delegation but requires (a) the delegated object to explicitly deal with an extra argument, and (b) cannot easily cope with new methods for the delegated object. In such cases, we consider it important to have a model that supports extensibility and reusability, and for these purposes provides mechanisms that may involve additional overhead.

- Third, it has been argued that some mechanisms in programming languages that incur a certain implementation overhead are required by the application anyway, and thus do not (or hardly) reduce the performance characteristics of the application. An example of this is inheritance and dynamic binding: the additional overhead of (in C++) the virtual function table that realises dynamic binding can probably be neglected, compared to the overhead that would be involved if the application programmer had to realise the same behaviour through additional coding.
- Fourth, some mechanisms, in particular those that increase the flexibility or dynamic adaptability of a system, incorporate an overhead that would not be needed in more static applications of the mechanism. When such cases can be identified, they can be optimised so that the generic mechanism requires just as much overhead as a limited version of the mechanism.

  An example of this is the dispatching mechanism in the composition-filters model, that supports dynamic inheritance. A straightforward implementation of the dispatch filter would assume this dynamic behaviour of objects at all times, involving a certain cost (i.e. conventional inheritance mechanisms would be more efficient). However, by considering the conditions in the dispatch filter, it can be deducted that in most cases these conditions will not change at all (i.e. they are `True`). In this case a more efficient implementation can be generated.

Concluding, the point that we make here is that seemingly inefficient language mechanisms are not always as inefficient as they look, or may well be worth some performance overhead. In particular, it is important to consider whether a mechanism is inherently inefficient, in all situations, or that simple and static cases can be optimised. Obviously, the language designer must still choose between expressive power and (potentially) reduced performance.

To include the opinion of performance-oriented researchers, we refer to the OOPSLA 1993 workshop 'Efficient Implementation of Concurrent Object-Oriented Programs'. Even though the common interest was in achieving maximum performance with the current software and hardware architectures, it was commonly agreed that ".. in the future, the focus may shift from efficiency to software productivity.." [Kale 94].

An additional point we would like to make is that the programmer does still carry some responsibility for potential loss of performance when certain mechanisms are applied. By providing methodological support for these situations, an explicit trade-off can be made during the design phase of application development.

An important property of composition filters for optimisation purposes is that it is a declarative mechanism, in the sense that the filter initialisation is a specification of certain properties, rather than an algorithm that implements certain behaviour. As a result, tailored or optimised implementations can be generated in most situations. It is much easier to derive object properties from a filter specification than it would be if these properties were embedded in fully expressive method implementations. This is exemplified in section 5.2, where a sequence of wait filter specifications is simplified into a single boolean expression in terms of the conditions of an object.

5.1.3 About this Chapter

In the subsequent sections of this chapter two implementation aspects of wait filters are addressed. The first aspect, covered by section 5.2, deals with reasoning about wait filters. It demonstrates how the acceptance of a message by a sequence of wait filters can be expressed by a boolean expression.

The second aspect that is treated in this chapter -in section 5.3- is the optimisation of the evaluation of conditions. A message that is blocked in the queue of an object must be activated when the conditions it depends on become true. In a naive approach this would require the re-evaluation of all the conditions after each state change of the object manager (according to the definition, only when the state of the object manager changes, an acceptable message needs to be dequeued, although this may also be done earlier). In section 5.3 it is shown how to avoid re-evaluations of conditions when the state of the object has not changed, and how to minimise the amount of computation involved in condition computation. The contents of section 5.3 have been described previously in [Bergmans 93].

In section 5.4 the architectural issues involved in the realisation of the composition-filters model are discussed based on a framework for merging Smalltalk-80 and the composition-filters model. Section 5.5 concludes this chapter.

5.2 Reasoning About Wait Filters

In this section we will discuss how we can reason about -sets of- wait filters. This can serve several purposes, one of these is the combination of multiple subsequent wait filters into a single synchronisation specification. This can for instance be used to inline synchronisation specifications in method bodies, or map wait filter specifications to method guards.

In addition to aspects of implementation, the techniques presented in this section can also be applied for different purposes. For instance to compare different synchronisation specifications and determine their equivalence. Another example is to detect messages that can never be accepted by the wait filters, i.e. the associated synchronisation constraint is always *false*.

5.2.1 The Abstract Syntax of Wait Filter Specifications

We use the abstract syntax of the composition-filters computation model defined in section 2.6 as the starting point of our discussion. To simplify the presentation, we strip this abstract syntax so that it represents a set of wait filters only. We also simplify the message processor part of filter specifications to deal only with matching of message selectors. As this is common for wait filter specifications this is not a severe restriction (there are no technical problems involved in including message substitution parts and target matching). Subsequently the synchronisation semantics of a set of wait filters are defined.

Assume we have a set of wait filters *WaitSet* with the following abstract syntax definition:

WaitSet ≝ *Filter** .
Filter ≝ *init:FiltElems* .
FiltElems ≝ *FiltElem** .
FiltElem ≝ *cond:Condition; operator:ExclOper; messPart:MessProcs* .
  Condition ≝ *selector:Identifier* . *// we omit details that are not used here*
  ExclOper ≝ *Enable | Exclusion* .
  MessProcs ≝ *MessProc** .
    MessProc ≝ *Identifier | Wild card* . *// simplified version of definition in section 2.6*
  Message ≝ *selector:Identifier* . *// we omit details that are not used here*

As an example a wait filter specifying synchronisation for a bounded buffer is expressed in terms of the abstract syntax. The synchronisation specification defines the synchronisation constraints of a bounded buffer object with mutual exclusion.

```
bufferSync : Wait = { Empty=>put, Partial=>{get, put}, Full=>get, True~>{get, put} };
mutexSync : Wait = { Free=>*, Recursive=>* };
```

This syntactical filter specification is expressed in terms of the abstract syntax as follows:

syncSpec ≝ *WaitSet(* <bufferSync, mutexSync> *).*
  bufferSync ≝ *Filter(init:FiltElems(* <el1, el2, el3, el4>*)).*
    el1 ≝ *FiltElem(cond:*Empty, *operator:Enable, messPart:MessProcs(*<"put"*>)).*
    el2 ≝ *FiltElem(cond:*Partial, *operator:Enable, messPart:MessProcs(*<"get", "put"*>)).*
    el3 ≝ *FiltElem(cond:*Full, *operator:Enable, messPart:MessProcs(*<"get"*>)).*

el4 $\stackrel{def}{=}$ *FiltElem(cond:*True, *operator:Exclusion, messPart:MessProcs(*<"get", "put">*))*.

mutexSync $\stackrel{def}{=}$ *Filter(init:FiltElems(* <el5, el6>*))*.

el5 $\stackrel{def}{=}$ *FiltElem(cond:*Free, *operator:Enable, messPart:MessProcs(<Wild card>))*.

el6 $\stackrel{def}{=}$ *FiltElem(cond:*Recursion, *operator:Enable, messPart:MessProcs(<Wild card>))*.

We assume that the definitions of the conditions Empty, Partial, Full, True, Free and Recursion as boolean expressions are available, and will restrict ourselves to referring to these conditions through their respective identifiers. Next we define meaning functions for the abstract syntax.

5.2.2 Deriving Synchronisation Constraints from Wait Filters

By defining meaning functions for the rules in the abstract syntax, we will demonstrate how simple synchronisation constraints can be derived for each message selector (cf. guards). The important property of this transformation is that it turns the filter specifications into a boolean expression. This is both more suitable for straightforward implementation and is easier to manipulate, evaluate and reason about.

WaitSet

The semantics of a set of wait filters is defined by offering a message to a set of wait filters, which returns a boolean value. This value is determined by a boolean expression in terms of the conditions that appear in the filters. The results of the respective filters in the set are combined with a boolean AND operation. Thus, a message must match with all the filters, or it cannot be accepted.

$WaitSet : Message \rightarrow \mathbb{B}$

$WaitSet \llbracket filters \rrbracket (mess) \stackrel{def}{=}$

over *filters* **apply** λ *filter* • *Filter* $\llbracket$ *filter* $\rrbracket$ *(mess)* **combine** $\wedge$ **empty** *true* **end**

Filter

This rule in the abstract syntax is derived from the original abstract syntax in 2.6, but has no function here.

$Filter : Message \rightarrow \mathbb{B}$

$Filter \llbracket$ <*init*> $\rrbracket$ *(mess)* $\stackrel{def}{=}$ *FiltElems* $\llbracket$ *init* $\rrbracket$ *(mess)*

FiltElems

The subsequent elements in a filter are combined through a boolean OR; when a message matches with any of the filter elements, it is accepted.

$FiltElems : Message \rightarrow \mathbb{B}$

$FiltElems \llbracket elems \rrbracket (mess) \stackrel{def}{=}$

over *elems* **apply** λ *elem* • *FiltElem* $\llbracket$ *elem* $\rrbracket$ *(mess)* **combine** $\vee$ **empty** *false* **end**

FiltElem

A filter element consists of three parts: a condition, an exclusion operator (either enable or exclude) and a message processor part. Depending on whether the message that is provided as an argument is accepted by the message processor part, and on the type of exclusion operator, that particular message is either never accepted, or it is accepted when the associated condition is true.

Two cases appear here: by looking at the exclusion operator and the message processor on the right hand side of the filter element, either the message can match with the filter element or it can never match. In the first case a match still depends on the condition of the filter element, which is then returned. In the second case *false* is returned.

The meaning function of the *ExclOper* takes the matching of the message as an argument, as determined by the meaning function *MessProcs*. Then either the result of the condition is returned, or the boolean value *false*:

FiltElem : Message → 𝔹

FiltElem ⟦ <*cond, oper, messpart*> ⟧ *(mess)* $\stackrel{def}{=}$
  if *ExclOper* ⟦ *oper* ⟧*(MessProcs*⟦ *messpart* ⟧*(mess))*
  then *Condition* ⟦*cond*⟧*(mess)*
  else *false* **end**

ExclOper

This is the operator that separates the condition from the message processing part. It can be either '=>', which is the enable operator, or '~>', which is the exclusion operator. A boolean argument *match* is provided that designates whether the message matched at the message processor part. If the operator is the enable operator, a match of the message will result in -conditional- acceptance, and in case the message does not match, it is rejected. If the operator is the exclusion operator, however, messages that match are never accepted, but only messages that do not match are accepted:

ExclOper: 𝔹 → 𝔹

ExclOper ⟦ *oper* ⟧*(match)* $\stackrel{def}{=}$
  case *oper* **of**
    Enable ⇒ *match* |
    Exclusion ⇒ ¬*match*
  end

MessProcs

This iterates over the elements in the list until one of the elements matches. If none of the elements matches *false* is returned.

MessProcs: Message → 𝔹

MessProcs ⟦ *procs* ⟧ *(mess)* $\stackrel{def}{=}$
  if *procs.EMPTY*
  then *false*
  else if *MessProc*⟦ *procs.HEAD* ⟧*(mess)*
      then *true*
      else *MessProcs*⟦ *procs.TAIL* ⟧*(mess)* **end**
  end

MessProc

This deals with the matching of messages. Two different situations are distinguished: the filter defines a wild card, in which case all messages will match. Otherwise the selector is tried to match against the selector of the message:

MessProc : Message → IB
MessProc ⟦ *selec* ⟧ *(mess)* $\stackrel{def}{=}$
case *selec* **of**
Wild card⇒ true |
Identifier⇒ selec=mess.selector
end.

Condition

We do not elaborate on the conditions here and simply substitute the condition identifier here:

Condition : Identifier
Condition ⟦ *cond* ⟧ *()* $\stackrel{def}{=}$ *cond.identifier.*

These semantics for a set of wait filters take a message and result in a boolean expression, consisting of logical ANDs and ORs, and conditions. This boolean expression defines the acceptance of that message.

In the bounded buffer synchronisation example, this results in the following expressions:

WaitSet ⟦ *syncSpec* ⟧ *(*getMessage*)* $\stackrel{def}{=}$ *(false* ∨ Partial ∨ Full ∨ *false)* ∧ (Free ∨ Recursive)
= (Partial ∨ Full) ∧ (Free ∨ Recursive)
where getMessage $\stackrel{def}{=}$ *Message(selector(identifier:*get*)).*

WaitSet ⟦ *syncSpec* ⟧ *(*putMessage*)* $\stackrel{def}{=}$ (Empty ∨ Partial ∨ *false* ∨ *false*) ∧ (Free ∨ Recursive)
= (Empty ∨ Partial) ∧ (Free ∨ Recursive)
where putMessage $\stackrel{def}{=}$ *Message(selector(identifier:*put*)).*

For all messages with a selector that is not equal to the put or get, the following constraints apply:

WaitSet ⟦ *syncSpec* ⟧ *(*otherMessages*)* $\stackrel{def}{=}$ *(false* ∨ *false* ∨ *false* ∨ True*)* ∧ (Free ∨ Recursive)
= True ∧ (Free ∨ Recursive) = Free ∨ Recursive.
where otherMessages $\stackrel{def}{=}$ *Message(selector(identifier:sel)).*
where (sel≠put*)* ∧ *(sel*≠get*).*

5.2.3 Equivalence of Synchronisation Specifications

We can use the transformation technique above also to determine whether two synchronisation specifications are equivalent. We refrain here from comparing different condition implementations; this is possible as well in many cases, but difficult in general because of the full expression power available for condition implementations.

As an example consider the following synchronisation specification:

```
mutexSync : Wait = { Free=>*, Recursive=>* };
syncSpec1 : Wait = {  {Partial, Full}=>get, True~>get };
syncSpec2 : Wait = {  {Empty, Partial}=>put, True~>put };
```

We will now show that this specification is equivalent to the synchronisation specification shown in the previous subsection.

The first step is to transform multiple conditions into separate filter elements for the filters syncSpec1 and syncSpec2:

```
{ {Partial, Full}=>get, True~>get }  ⇔ { Partial=>get, Full=>get, True~>get };
{ {Empty, Partial}=>put, True~>put }  ⇔ { Empty=>put, Partial=>put, True~>put };
```

Then we can apply the transformation algorithm to the resulting set of three wait filters, this results in the following expressions:

WaitSet ⟦ *syncSpec* ⟧ *(*getMessage*)* $\stackrel{def}{=}$
(Free ∨ Recursive) ∧ (Partial ∨ Full ∨ *false*) ∧ (*false* ∨ *false* ∨ True)
= (Free ∨ Recursive) ∧ (Partial ∨ Full) ∧ True = (Partial ∨ Full) ∧ (Free ∨ Recursive)
where getMessage $\stackrel{def}{=}$ *Message(selector(identifier:*get*)).*

WaitSet ⟦ *syncSpec* ⟧ *(*putMessage*)* $\stackrel{def}{=}$
(Free ∨ Recursive) ∧ (*false* ∨ *false* ∨ True) ∧ (Empty ∨ Partial ∨ *false*)
= (Free ∨ Recursive) ∧ True ∧ (Empty ∨ Partial)=(Empty ∨ Partial) ∧ (Free ∨ Recursive)
where putMessage $\stackrel{def}{=}$ *Message(selector(identifier:*put*)).*

For all messages with a selector that is not equal to the put or get, the following constraints apply:

WaitSet ⟦ *syncSpec* ⟧ *(*otherMessages*)* $\stackrel{def}{=}$
(Free ∨ Recursive) ∧ *(false* ∨ *false* ∨ True*)* ∧ *(false* ∨ *false* ∨ True*)*
= (Free ∨ Recursive) ∧ True ∧ True = Free ∨ Recursive.
where otherMessages $\stackrel{def}{=}$ *Message(selector(identifier:sel)).*
where (sel≠put) ∧ *(sel≠get).*

The importance of the transformation algorithm is that it maps filter expressions to boolean expressions. The characteristics of the boolean operators AND and OR can then be used for manipulating and simplifying the synchronisation expressions.

5.3 Condition Evaluation

In this section we discuss the evaluation of conditions. The conditions must be re-evaluated repeatedly in order to provide timely synchronisation as required by changes in the object state. This potentially involves a significant performance overhead. We propose a technique to reduce this overhead, without prescribing a particular implementation for conditions.

In this section first the problem is defined, the approach that we have taken is explained. Then the algorithm is described and the technique is illustrated with an example. We end with some concluding remarks. The technique that is presented here was first described in [Bergmans 93].

5.3.1 Problem Statement & Approach

Our choice of adopting arbitrary message expressions for specifying conditions may have significant influence on the performance of the application. The reason for this is that the mechanism embodies the concept of repeated condition (re-)evaluation: after each change in the state of the object manager, all conditions must be evaluated again. Obviously, implementing this in a straightforward manner by indeed executing the bodies of all the conditions after each state change of the object manager incurs a serious amount of overhead.

In order to reduce the amount of overhead, we set two goals: (1) to reduce the frequency of condition evaluation, and (2) to minimise the computational cost involved in the evaluation.

Two ideas form the basis of our approach: first, by finding the locations where the relevant state changes take place we can avoid (redundant) continuous evaluation. The second objective is to minimise the amount of condition evaluation code (and thus the computational expense) by tailoring the code to the specific state change and context. We will now briefly explain how this is achieved. The algorithm is applied while compiling a single object, though better results can be obtained if the source-code of the complete application is available.

For every condition defined by the object, a dependency-graph is constructed, which is similar to a parse-tree. The nodes in this tree are (simple) message expressions of the form receiver.mess(args). A re-evaluation of a condition is initiated when (local) code is executed that *conflicts* with the condition. Condition re-evaluation code is generated and inserted after the conflicting code in the method. The re-evaluation code is derived from the dependency graph by considering only the subset of the graph that is relevant for the particular conflict(s).

A piece of code is said to conflict with a condition, when the *read-set* of the condition overlaps with the *write-set* of the code. Intuitively, this means that the code affects the state of one or more objects that the condition depends on. The read-set of a code-block (either a condition or a method) is the set of objects that is 'read', i.e. that read(-only) messages are sent to. Similarly, the write-set is the set of objects that a block of code sends one or more write messages to. The read/write behaviour of a method can be determined (recursively)

from the read/write behaviour of the method it calls in turn. Read/write behaviour is known for all system-defined ('primitive') objects.

The optimisation algorithm is partitioned into the following phases:

1. The creation of dependency graphs for conditions.
2. The construction of read-sets for every condition.
3. The construction of write-sets for every method.
4. Detecting conflicts between conditions and method code.
5. Generating and inserting condition evaluation code for every method.

5.3.2 The Algorithm

We will now discuss each of the five phases separately. Although the discussion is quite detailed, there are still a large number of issues that are ignored or not touched upon in this discussion. A number of these are discussed at the end of this subsection.

1. Construction of Dependency-Graphs

A condition is a message expression having the following format:

messExpr ::= *receiver* selector *argument*.*
receiver ::= *object.*
argument ::= *object.*
object ::= constant | variable | *messExpr*.

A (rather complex) example of this is:

condition::= o1.m1(o2.m2(o3, o4).m3.m4(o5.m5)).m6(o6, o7.m7, o8.m8(o9.m9)).m10

It is possible to make a transformation of this structure into another structure, consisting of simple assignments only. This new structure is expressed by the following rules:

messExpr ::= *assign*.*
assign ::= variable ':=' *receiver* selector *argument*.*
receiver ::= *object.*
argument ::= *object.*
object ::= constant | variable.

argument and receiver objects are no message expressions in this structure, but can only be variables that contain the result of a message expression. Thus a number of new -temporary- variables are introduced. For example, the condition example above can be transformed into:

```
t9 := o2.m2(o3,o4) ;
t8 := o5.m5 ;
t7 := t9.m3 ;
t6 := o9.m9 ;
t5 := t7.m4(t8) ;
t4 := o8.m8(t6) ;
t3 := o7.m7 ;
t2 := o1.m1(t5) ;
t1 := t2.m6(o6, t3, t4) ;
condition := t1.m10 ;
```

We can construct a tree that represents this condition. Every node in the tree represents a single message expression. The "□" symbols represent -unnamed- temporary variables. The tree is shown in figure 5.3.1.

This tree can also be regarded as a dependency graph: every node in the tree depends on its child nodes. Thus, when some of the objects o*i* that appear in the expression-tree are not changed, it is not necessary to re-compute the complete tree. A node *n* needs to be re-evaluated when its expression contains modified objects. In addition, the nodes that depend on *n* need to be re-evaluated. These are the nodes that are encountered when following the path from *n* to the root of the tree.

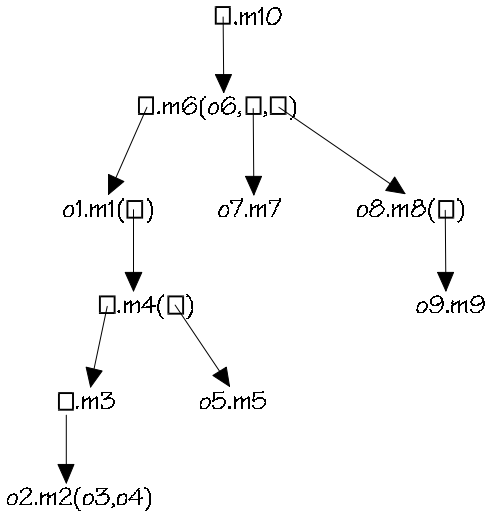


Figure 5.3.1 The dependency tree

We will use this dependency information in phase 5 to minimise the amount of re-evaluation code.

2. Construction of Read-sets

For detecting possible conflicts a read-set must be determined for every condition. The read-set is the set of objects that the condition refers to. Due to the rule that a condition is side-effect free, all methods that are, directly or indirectly, called by a condition are read-only methods[1].

To determine its read-set, the condition expression must be parsed, to find all the message expressions of the form "receiver.mess(args..)". In order to proceed, first the class of receiver is to be determined as accurately as possible. Although preferable, it is not necessary to know the precise class. It is sufficient to determine a set of classes with the guarantee that the actual class of receiver during execution must always be contained in the set. In the subsequent discussion we will usually refer to *the* class of an object, where we actually mean the set of classes. In some cases no assumptions can be made at all about the class of the receiver; then it is tagged as *undependable*.

The information about the class of the receiver is used to determine the set of objects that are read while executing an expression "receiver.mess(args..)". If receiver is a message expression by itself, it will be parsed first, meanwhile maintaining the read-set, and a result type will be determined. Note that the arguments that are provided by the message

---

1 The algorithm for determining the read-set can also be used to detect violations of this rule, although due to dynamic binding, not *every* violation can be detected statically.

invocation can be read or written as well. This recursive process continues until the receiver is a primitive object (an object with an implementation that is provided by the system).

We do not elaborate here on how to determine the class(es) of the receiver. Various approaches are possible, e.g. the *type inference* algorithm in [Palsberg 91] does exactly what we need here, for a completely untyped language. Since our implementation language *Sina* is typed[2], in most cases the class of an object can be determined more easily.

By considering the class of the receiver object, the object can be classified as one of these three kinds:

- *primitive object*: for every method of a primitive object, it is known whether it reads and/or writes the state of the primitive object and the method arguments. Based on this information, the receiver object and the arguments can be added to the read-set.
- *user-defined object*: this means that the object is an instance of a user-defined class, of which the source-code must be available. Then the read-set of `mess` is to be determined (this is a recursive process), and will be added to the read-set of the condition. In the case that no source code and further information is available, the object is tagged as undependable.
- *undependable object*: for some receiver objects no static assumptions can be made, usually due to dynamic binding. For example the values and classes of pseudo-variables such as *server* (comparable to Smalltalk *self*) and *sender* (refers to the sender of the message) can sometimes only be known at run-time. Objects are also marked as undependable when they may be modified implicitly by other processes. For undependable objects worst-case behaviour is assumed; they are always included in the read-set (and in the write-set as well), and the arguments that are supplied in the message invocation as well.

3. Construction of Write-sets

In this phase, the methods of the object are examined to determine the write-sets of all message expressions. The aim of determining write-sets is to find as accurately as possible the locations in the code that affect the state of the object, and thus indirectly the state of the conditions for that object.

A method body consists of a sequence of message expressions or assignments:

body ::= *expr**.
expr ::= *assignment* | *messExpr*.
assignment ::= variable ':=' *messExpr*.
messExpr ::= *receiver* selector *argument**.
receiver ::= *object*.
argument ::= *object*.
object ::= constant | variable | *messExpr*.

Each of the expressions is parsed to construct a write-set. This is done similarly to the way read-sets are constructed. However, when an assignment is encountered, the variable on the

[2] Although we allow to specify a type to be *Any*, which basically means that the variable is no longer type-checked statically.

left-hand side is added to the write-set, and primitive objects are only added to the write-set if the message[3] that is sent to it is specified to be a write-message.

The result of this phase is the association of the write-set with each message expression within the method or with each method (e.g. for mutual exclusive objects).

4. Detection of Conflicts

The detection of conflicts between conditions and methods is used to locate the places where the execution of method code may directly cause a condition to change from true to false or reverse. This may occur when the objects or values that a method writes to (i.e. modifies) appear in the expression that defines a condition. This is the case if the write-set of an expression overlaps with the read-set of a condition (or: the intersection of the two sets is non-empty).

The detection of conflicts serves two purposes: (a) for determining the locations where the condition re-evaluations can be inserted in the method code. In this case for each statement in a method the write-set must be determined, though. And (b) to generate tailored condition re-evaluation code, by considering which objects are involved in the conflict.

5. Generation of Condition Evaluation Code

The dependency graph constructed in phase 1 is now used to tailor (i.e. minimise) the amount of code that is to be executed for the re-evaluation of conditions. This is achieved by considering only the part of the dependency tree that really needs to be executed again. These are the nodes whose result is depending on an object that has (presumably) changed. These nodes can be found by taking the path from the changed object up to the root of the tree.

When expressing the condition as a sequence of assignments, as was shown during phase 1, the temporaries that are introduced can be used for caching of intermediate results. Thus, for every condition a 'cache' for intermediate results is reserved. It is possible to avoid redundant caching, for example of values near the root of the tree that are always recomputed. This can be achieved by first marking all the nodes in the tree that are entry points for re-evaluation, and 'inlining' the other nodes of the tree.

There are several alternative strategies for exploiting the dependency graph and the caching of intermediate results:

- *Code generation*: the dependency graph can be used for generating optimised re-evaluation code: when detecting a conflict between a method and a condition on a particular object, the node[4] in the dependency graph that refers to that object is located. From the path from that node to the root of the tree a re-evaluation expression is constructed, which uses the cached values for sub-trees that need not be re-computed. Obviously, the cached values should be up-to-date. The code for the evaluation of the

---

3 Strictly, when the method of the primitive class that corresponds to the message that is sent to the instance of the primitive class is defined to be a write method.

4 There can actually be multiple nodes referring to the same object; this means that there are multiple paths that have to be combined. We will further ignore this.

sub-tree can be inlined into the method that caused the state change. Whenever, possible, we take this approach, as it is the most efficient.

- *Run-time tree traversal*: a run-time evaluation mechanism can be defined that starts the evaluation at a particular node in the tree (some representation of the tree needs to be available). This evaluation must be triggered by the -suspected- change of the state of one of the objects in the tree. This approach can be useful when adopting a trigger mechanism for *undependable* objects; these objects initiate re-evaluation themselves, but this cannot be inlined.
- *Hybrid*: code can be generated for each possible re-evaluation entry point: when there are *n* objects the condition depends on, this will result in *n* generated evaluation expressions. Then at run-time, hooks to these entry-points can be handed out (in combination with a trigger mechanism). Again, for the generated evaluation expression better optimised code can be generated. A space/performance trade-off must be made to determine whether this approach is useful.

As a further optimisation, it is possible to collect all conflicts arising in the body of a method, and insert the condition re-evaluation code at the end of the method body. This is especially useful when the presence of *undependable* objects would otherwise (without further measures) result in condition re-evaluation after every statement of the method. However, it should be noted that this strategy may be visible to the application programmer in certain cases. For mutual exclusive objects it can be applied without causing problems.

Another consideration may be to inline very simple expressions instead of cache the intermediate results. This may be more efficient, as it does not require space to be reserved for caching results, and storing and retrieving those results. Compiler heuristics should decide when to do caching of intermediate results, and when to do complete re-computations.

Additional Aspects

In the previous discussion we did not address the following issues:

- Conditions frequently consider typical synchronisation information, such as the number of executing threads, the number of messages that are waiting in the queue, etc. Such information is maintained by, and can be retrieved from, the *object manager* of an object. A dummy object representing the object manager can be inserted as an instance variable during the execution of the algorithm. Updating and retrieval of synchronisation information is modelled by inserting message sends to this dummy object at the appropriate places.
- We have assumed that only the local methods cause the state change that the condition depends on. This is certainly not always the case for *external* (or global) objects, and for objects that contain internal processes. Such situations are taken into consideration by marking such objects as *undependable*. A similar approach is taken for objects that may be overridden dynamically with other (subtype-) objects.
- Special attention is needed to take care of the following issues: pseudo-variables (*self, server, sender* and *message*), inherited and delegated methods, and recursive calls.

- The actual acceptance and activation of messages is not discussed here; this requires the insertion of additional code that takes into account the modified state of conditions to determine message acceptance.
- The algorithm as we described it can be characterised as performing *global optimisation*, in the sense that the source code of all the classes in the application must be available. *Incremental optimisation* is also possible: it determines read- and write-behaviour on per-class basis. This means that the context in which a class is used is not taken into account, thus reducing the amount of optimisation that is possible. But the source code of the accessed classes is no longer required, and incremental optimisation greatly reduces the amount of code that is traversed to determine read/write behaviour.
- We did not address the problem of concurrency during condition evaluation here. If the acceptance of a message requires multiple conditions to be true at the same time, and message acceptance is done concurrently with condition re-evaluation, inconsistencies might occur. A solution is to -atomically- make a copy of the states of all the conditions before determining the synchronisation of a message. This avoids that message synchronisation is based on inconsistent condition states.

5.3.3 An Example

To demonstrate this technique we use the bounded buffer example again, including some of the behaviour that is usually added implicitly by default. In particular this is the synchronisation filter that realises mutual exclusion (mutexSync) and the conditions that are used by this filter.

The definition of this class is as follows:

```
class BoundedBuffer(limit:Integer) interface
    comment implements a bounded buffer with synchronisation;
    conditions
        Empty; Partial; Full;
    methods
        put(Any) returns Nil;
        get returns Any;
        peek returns Any
    inputfilters
        bufferSync : Wait = { Empty=>put, Partial=>{get, put}, Full=>get, True~>{get, put} };
        mutexSync : Wait = { Free=>*, Recursive=>* };  // this is usually added by default
        disp : Dispatch = {inner.* };
end // class BoundedBuffer interface

class BoundedBuffer implementation
    instvars
        store : Array(limit);  // index ranges from 1 to <limit>
        head, tail : Integer;
    conditions
        Empty  begin return head=tail end;  // no elements in the buffer
        Partial  begin return (inner.Empty.not and inner.Full.not) end;
        Full  begin return (limit+tail-head).mod(limit)=1;  end;  // the buffer is full
        Free  begin return ^self.active=0 end ;
        Recursive  begin  return message.isRecursive;  end;
    initial
        begin head:=1; tail:=1; end;
```

```
    methods
        put(elem:Any) returns Nil
            begin store.atPut(head, elem); head:=head.mod(limit)+1; end;
        get returns Any
            begin return store.at(tail); tail:=tail.mod(limit)+1; end;
        peek returns Any
            begin  return store.at(tail);  end;
        // other methods, e.g. for allowing direct access to the buffer by subclasses
end // class BoundedBuffer implementation
```

Phase 1: The dependency graphs for the conditions are formed by the following restructured condition implementations:

```
Empty = head.equals(tail);
Partial = t1.and(t2); t1=t3.not; t3=inner.Empty; t2=t4.not; t4=inner.Full;
Full = t1.equals(1); t1=t2.mod(limit); t2=t3.minus(head); t3=limit.add(tail);
Free = t1.equals(0); t1=^self.active;
Recursive = message.isRecursive;
```

Phase 2: The read-sets of the conditions are then determined, to do this some inlining is required (for local conditions and methods this is always possible). This gives the following read-sets for each condition:

```
Empty : { head, tail }
Partial : { head, tail, limit }     // the objects referenced by other conditions included
Full : { limit, tail, head }
Free : { ^self }    // this means that the condition depends on the state of the object manager
Recursive : { message }    // for every newly arrived message, this condition can be changed
```

Phase 3: The write-sets of the methods are determined in a similar way. We assume that the state of the object manager and of the pseudo-variable `message` is taken care of separately.

```
initial : {head, tail }
put : { store, head }   // the characteristics of primitive class Array are known
get : { tail }
peek : ∅
```

Phase 4: Detection of conflicts for each method. Because in this example the object is mutual exclusive, we determine conflicts on a per-method basis only. For each method, we compare the write-set with the read-sets of all the conditions:

```
initial :
    initial ∩ Empty = {head, tail}
    initial ∩ Partial = {head, tail}
    initial ∩ Full = {head, tail}
    initial ∩ Free = ∅
    initial ∩ Recursive = ∅
put :
    put ∩ Empty = {head}
    put ∩ Partial = {head}
    put ∩ Full = {head}
    put ∩ Free = ∅
    put ∩ Recursive = ∅
```

```
get :
    get ∩ Empty = {tail}
    get ∩ Partial = {tail}
    get ∩ Full = {tail}
    get ∩ Free = ∅
    get ∩ Recursive = ∅
peek :
    peek ∩ Empty = ∅
    peek ∩ Partial = ∅
    peek ∩ Full = ∅
    peek ∩ Free = ∅
    peek ∩ Recursive = ∅
```

If the intersection of the write-set of a method and the read-set of a condition is empty, then the condition will never be affected by the execution of the method. For example, the peek method does not change the state of the object at all, and therefore does not require the re-evaluation of conditions. As the object manager and pseudo-variables is not dealt with within the bodies of the methods, the Free and Recursive conditions never conflict here.

Phase 5: Generation of condition evaluation code. In this example, we insert the following condition evaluation code at the end of each method. At least for all the conditions a variable is reserved to maintain (cache) their states.

```
initial:
    Empty := head.equals(tail);
    full3:=limit.add(tail); full2:=full3.minus(head); full1:=full2.mod(limit);
    Full:=full1.equals(1);
    partial1:=Empty.not; partial2:=Full.not; Partial:=partial1.and(partial2);
```

Because the initial method initialises both the head and tail, all conditions must be evaluated. Note however, that if subsequently messages arrive that do not modify the state of the instance variables (e.g. get messages that are blocked or peek messages), the conditions Empty, Full and Partial are *not* re-evaluated.

```
put :
    Empty := head.equals(tail);
    full3:=limit.add(tail); full2:=full3.minus(head); full1:=full2.mod(limit); Full:=full1.equals(1);
    partial1:=Empty.not; partial2:=Full.not; Partial:=partial1.and(partial2);
```

The put method only affects the value of the head instance variable, therefore it is not necessary to compute the value of the temporary variable full3 again. We will later remove such redundant code from the evaluation expressions.

```
get:
    Empty := head.equals(tail);
    full3:=limit.add(tail); full2:=full3.minus(head); full1:=full2.mod(limit); Full:=full1.equals(1);
    partial1:=Empty.not; partial2:=Full.not; Partial:=partial1.and(partial2);
```

After the execution of a get method the conditions must be fully re-evaluated. Note however, that the values of the conditions are cached, which always avoids recomputation of the Full and Empty conditions when computing the value of condition Partial.

```
peek:
    // no re-evaluation code is added to the body of the peek method at all
```

Because the peek method has no conflicts, no re-evaluation code needs to be generated.

The code that we showed here can be optimised by removing redundant computation and unnecessary temporary variables. This results in the following code that must be added to the implementation of method bodies:

```
initial:
    Empty := head.equals(tail);
    full3:=limit.add(tail);
    Full:=full3.minus(head).mod(limit).equals(1);
    Partial:=Empty.not.and(Full.not);
put :
    Empty := head.equals(tail);
    Full:=full3.minus(head).mod(limit).equals(1);     // here we minimise recomputations
    Partial:=Empty.not.and(Full.not);
get:
    Empty := head.equals(tail);
    full3:=limit.add(tail);
    Full:=full3.minus(head).mod(limit).equals(1);
    Partial:=Empty.not.and(Full.not);
peek:
    // no re-evaluation code is added to the body of the peek method at all
```

A number of remarks can be made about this example. Firstly, the bounded buffer example was chosen because it is well-understood, and has been used as a running example throughout this thesis. It does not demonstrate fully the potential optimisations that can be achieved with the technique presented here, though. Secondly, this particular implementation of the bounded buffer is by no means the most efficient one. Again, this is because we found this more illustrative for demonstrating certain properties of conditions in previous chapters where we used the same example.

Finally, we want to emphasise that this example does demonstrate some important benefits of the optimisation technique. The primary goal of the technique we propose here is to minimise the amount of condition re-evaluations, with the 'side-effect' of optimising the re-evaluation code itself through avoiding redundant computations. In particular the amount of re-evaluations of conditions that depend on the value of instance variables only is minimised.

In this example this can be observed for the conditions Empty, Partial and Full. Instead of re-evaluating these conditions repeatedly, they are only computed after the execution of the initial method, the get and the put method. If a lot of messages such as peek, equals and print are sent to the object, this will not require these conditions to re-evaluated at all. A straightforward implementation would require this after each change in the state of the object manager, i.e. message reception, method execution, method termination, etcetera.

5.3.4 Conclusions

We have shown a technique for reducing the amount of overhead involved in the evaluation of conditions. In particular if conditions and the objects they are defined for are static and self-sufficient minimised condition re-evaluation code will be inserted only at the points where state changes are made that affect the conditions.

We summarise the following aspects:

- ❏ The technique is applicable for state abstraction in objects that support intra-object concurrency.
- ❏ Only at those locations in method implementations where relevant state changes occur, re-evaluation code needs to be inserted .

- Re-evaluation code is minimised by employing a caching mechanism for intermediate results.
- The approach is suitable for both global (i.e. for all classes of the application the source code is available) and incremental optimisation and can cope with dynamic binding. Global optimisation can in many cases statically determine the exact nature of arguments and dynamically bound pseudo-variables, which allows for better optimisations.
- There is no performance 'overhead' involved in applying this technique. So even if no minimisation of condition evaluations or code reduction can be achieved, there is no performance penalty. However, if an object has a lot of complex conditions, but receives almost only messages that require few condition evaluations, an on-demand approach to condition evaluation may be more efficient.

The technique will usually involve some additional costs though; the first is the duplication of code because -tailored- versions of the condition implementations are inserted into the method bodies. The second is the requirement of additional memory for caching intermediate results and the states of the conditions. The latter are booleans only, but intermediate results in condition expressions might be complex objects. This will probably be exceptions, though.

Additional tuning of the proposed mechanism may thus be necessary or desirable; in several cases a space/performance trade-off has to be made. Note that the presented optimisations are independent from conventional program optimisation techniques, such as constant propagation, or elimination of redundancies. Such techniques can be applied in a subsequent phase. For instance in the bounded buffer example, if condition re-evaluation code is added to the body of the initial method, constant folding can reduce the generated code significantly.

An interesting aspect of this optimisation technique is that it offers a compromise between message guards and behavioural abstractions. For instance, in [Matsuoka 93b] it is remarked that the guard approach leads to a lot of -redundant- guard evaluations when a large number of messages is in the queue of the object. Behavioural abstractions, on the other hand, must predict the state of the object when the message is evaluated, which is not always possible. Our approach to synchronisation is more related to method guards, but the optimisation technique in fact results in a computation that is similar to behavioural abstractions: within the bodies of methods the behavioural abstraction is computed, which can then be used directly for synchronisation purposes without additional evaluations. However, we are able to deal with intra-object concurrency.

The problem of efficiently implementing conditions is more general than only for the implementation of the synchronisation mechanism in *Sina*. First, the synchronisation mechanism we proposed can be applied, with some modifications, to other object-oriented concurrent languages. Second, conditions provide, in essence, a mechanism for detecting when an object is in a particular state. This is also useful in other domains, such as for the generation of events and exceptions, or the detection of constraint violations.

5.4 The Sina Framework in Smalltalk

To test the examples in this thesis, a framework for integrating composition filters with the Smalltalk system was developed. This framework is briefly described in this section. This reveals some important architectural issues involved in the implementation of the composition filters object model.

An essential technique for implementing the composition filters computation model is reflection on messages; a message that is received by an object may be tested on various characteristics, such as the message selector, the arguments or the sender of the message, and modified by filters. This means that a received message must be *reified* to a first-class representation that can be manipulated by the filters and activated into a message invocation again[1]. We will show how this can be realised on top of the Smalltalk-80 system such that it is transparent to the calling -Smalltalk- object.

The framework that is discussed in this section aims at integrating Smalltalk with the composition-filters computation model; this means that a Smalltalk object can be extended with one or more composition-filters. This creates two types of objects in the system: pure Smalltalk objects, and objects with composition filters. To simplify the discussion, the latter will be termed as *Sina objects* and the first as *Smalltalk objects*[2]. Similarly, a Sina object can have two kinds of methods; *Sina methods* are only invoked after passing the filters of the object, and *Smalltalk methods* can be called directly by another object, without passing any filters.

As the framework involves no compilation or pre-processing phase, the filters are essentially interpreted during message evaluation[3]. The body of the methods of a Sina object consists of pure Smalltalk statements. Instance variables and message arguments can be both Smalltalk or Sina objects. This means that the performance of application code is not reduced because of unnecessary filtering overhead; only if composition-filter functionality is needed, some additional overhead is involved.

Message Reification

Figure 5.4.1 illustrates how filters are integrated within Smalltalk, and in particular how message reification is achieved[4]. Each Sina object inherits from class SinaObject, which defines the common behaviour of Sina objects, in particular for filtering messages. An important distinction between Smalltalk methods and Sina methods is that the names of the

---

1 This is also the essential functionality of Meta filters.

2 Whereas in fact all are essentially Smalltalk objects.

3 In fact, during object initialisation, the syntactical filter specifications are parsed and converted into a more convenient and efficient data structure for run-time evaluation. This does not involve significant optimisations or code generation.

4 The actual reification is performed by the Smalltalk system itself. If this were not the case (for instance in C++ this cannot be done) it would not be possible to add composition filters to an object without the clients of the object being aware.

latter are extended with the tag 'SINA'. Thus, the method m defined by object O2 is declared as SINAm. As a result, when invoking a message with the selector m on object O2, the appropriate method will not be found.

If a message is sent to an object (① in the figure) that does not support that message (including inherited methods from superclasses), the Smalltalk system invokes the message doesNotUnderstand: on the receiver object (②). This is in fact a reification of the message, as a first-class representation of the actual message that was sent is provided as an argument of the doesNotUnderstand: message. This reification is used to intercept the messages that are sent to an object, and let them be evaluated by the filters of the object. The doesNotUnderstand: method is (re-)defined in class SinaObject.

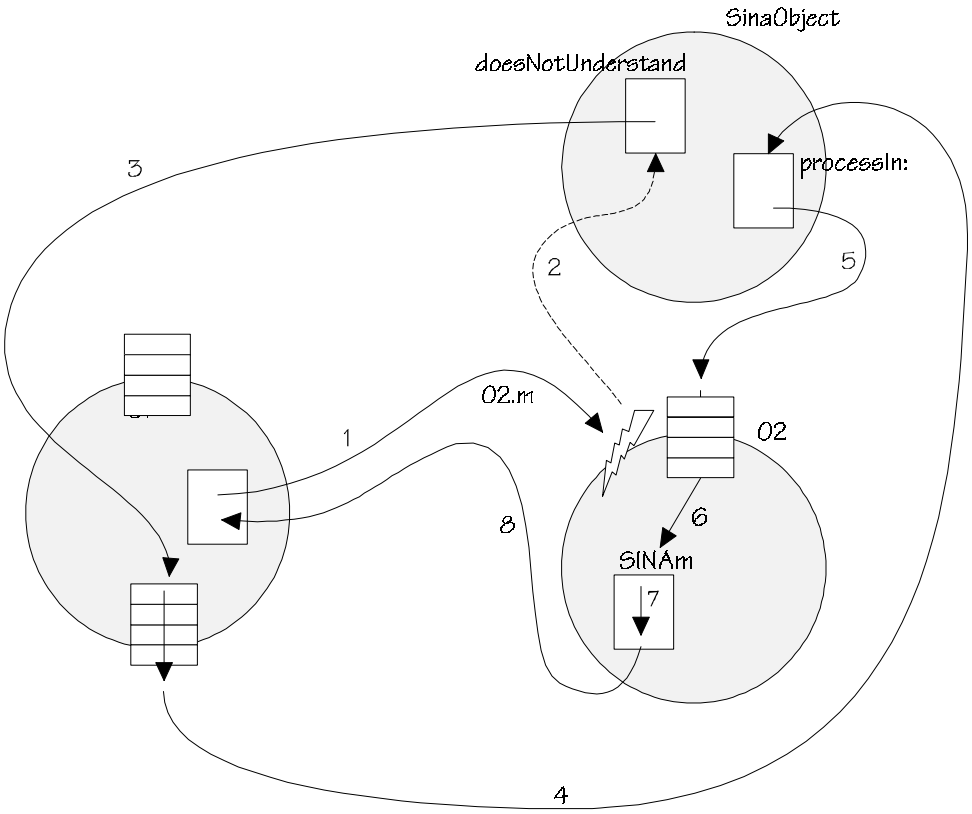


Figure 5.4.1 The flow of a message that is intercepted by filters.

However, before the input filters of the receiver object are evaluated, the output filters of the sending object are to be evaluated yet. This is taken care of by the doesNotUnderstand: method: it tests whether the sender of the message provides any output filters. Obviously, if the sender object is a Smalltalk object, it will not have any output filters at all. If the sender object does have output filters, they are evaluated, providing the reified message as an argument[5]. This is shown by arrow ③ in figure 5.4.1.

[5] In, fact, this shows a limitation of the mixing of Smalltalk and Sina objects: only messages that are sent to a Sina object are intercepted through the doesNotUnderstand: method, and thus only these messages will be evaluated by the outputfilters of the sender object.

If the outputfilters of object o1 do not redirect the message, the message is offered again to object O2 (arrow ④). The method processIn: handles such -reified- messages, it is also called directly by doesNotUnderstand: if the sender object does not have any output filters. The next step is to evaluate the filters that are defined by the receiver object O2 one by one (⑤). In the figure this evaluation results in the dispatch to a local method m. The dispatch filter extends the selector of the received message with the prefix 'SINA', and invokes the method with the corresponding selector (⑥), in the example this is SINAm. After the execution (⑦) of the method, the result of the method is returned (⑧) to the method of O1 that initiated the call.

First-class Message Representation

The crucial issue in this flow of control is the reification of the Smalltalk message invocation to a data structure that represents that message. The information about the message is partly generated by the Smalltalk system; this provides the message selector, the receiver object and the arguments. This first-class message representation is extended in our system with an additional data structure. This consists of a server variable, a sender variable, and a semaphore.

The server variable of a reified message is initialised with the original receiver of the message. Subsequent dispatches by the dispatch filter do affect receiver, but do not modify the value of the server variable. The variable sender is only initialised immediately after the reification as well. Its value is determined by accessing the context of the call (as provided by the Smalltalk system). The semaphore that is provided for each message is used to block the thread associated with the message and activate it again later.

Object Manager

The object manager maintains the state of the object. At a number of events, the object manager is informed of a change in the state of the object. The reception, blocking, activation or dispatch of a message and the start and termination of method executions are all signalled to the object manager, which adapts the various synchronisation counters accordingly. The object manager is also responsible for queuing and scheduling of messages that are blocked by wait filters.

Scheduling is non-preemptive in the current framework; this means that each thread is responsible for rescheduling, by invoking a so-called yield operation. Sina objects perform a rescheduling for every message send. This means that a method may claim all processor time until it makes a Sina message invocation. A method body with only pure Smalltalk code may thus cause starvation of other threads.

The synchronisation of incoming messages is, in this Smalltalk framework implementation, achieved by executing the following steps (and similarly for output filters):

1. A received message is reified and offered to the input filters. The input filters are evaluated by the calling thread.
2. When a message arrives at (a group of subsequent) wait filters, the message is put at the back of the message queue (i.e. the one associated with these wait filters).
3. The semaphore associated with the message is set to 0.
4. A snapshot is made from the values of the conditions.

5. The message queue of the object is searched from start to end to find a message that can be unblocked. For the first one that is encountered the associated semaphore is signalled, and no more messages are checked.
6. The thread performs a wait on the semaphore that is associated with the current message (i.e. the one that was just received). This may cause the thread to be blocked, and another thread to continue, or -if the semaphore was just signalled- the thread may immediately pass the semaphore.
7. The message is removed from the queue, and the next (non-Wait) filter is evaluated.

If a message is blocked, it can be rescheduled (by signalling the semaphore as in step 5) whenever the object manager is notified of a state change. This means that this implementation cannot deal with state changes that are not local, but are due to a state change in a nested or external object. A mechanism based on the notification of state changes to dependent objects has been implemented and tested in [Rein 94].

5.5 Discussion

In this chapter we discussed a number of issues involved in the implementation of Wait filters. In particular we focused on the optimisation of the evaluation of wait filters. This has been addressed in two steps; the first step is concerned with reasoning about wait filters. The second step deals with the evaluation of conditions. It attempts to determine as precisely as possible *when* a condition needs to be re-evaluated, and to *minimise* the amount of evaluation code through caching of partial results.

For reasoning about wait filters a translational semantics is defined that maps a set of wait filters to a function which expresses the synchronisation constraints of a message as a boolean expression in terms of condition values. The resulting mathematical specification of synchronisation constraints can be used for realisation or optimisation purposes, e.g. for constructing method guards, or for simplifying the synchronisation expressions. Another application is that the equivalence of synchronisation constraints can be determined. The key issue is that we port the synchronisation specifications to a well-known mathematical domain (i.e. that of boolean expressions) that better supports manipulation of and reasoning about the synchronisation constraints.

The optimisation of condition evaluations is achieved through a technique for reducing the amount of overhead involved in the evaluation of conditions. In particular if conditions and the objects they are defined for are static and self-sufficient minimised condition re-evaluation code will be inserted only at the points where state changes are made that affect the conditions.

Important properties are that the technique is applicable for implementing state abstractions in objects that support intra-object concurrency, and that the approach is suitable both for global and for incremental optimisation and can cope with dynamic binding. Global optimisation can in many cases statically determine the exact nature of arguments and dynamically bound pseudo-variables, which allows for better optimisations. Intermediate forms, for example when only the source code of application classes, but not of libraries is available are possible as well. However, if less information is available, more -probably redundant- condition evaluations are required.

The optimisation techniques that have been proposed in this chapter are only related to the semantics of wait filters and the evaluation of conditions. A full implementation of the composition-filters computation model involves numerous other aspects. Existing work on the implementation of object-oriented languages includes the Deutsch-Schiffman techniques [Deutsch 84] and the Hurricane [Atkinson 86], TS [Johnson 88b] and QUICKTALK [Ballard 86] optimising compilers for Smalltalk (dialects). Other optimisation techniques have been coined for the SELF system (e.g. [Chambers 92], [Hölzle 91]). But conventional optimisation and code generation techniques are often still applicable.

Evaluation

To test the optimisations that were presented in this chapter, we made some preliminary benchmarks, adopting the bounded buffer class that was used as an example in this chapter. The test application created a single bounded buffer with space for *b* elements[1]. Then four concurrent activities were created, which all competed for accessing the buffer. Two processes each perform *n* put operations, one process performs *2n* peek operations, and one process performs *2n* get operations.

The benchmarks measured the total time required for this application to execute, up to the termination of the last process. Variations were made for the number of operations (*n*) and for the size of the buffer (*b*). We tested two implementations of the bounded buffer; an unoptimised version and an optimised version that incorporated the techniques that were introduced in this chapter. The only difference between these two versions is that the unoptimised version evaluates messages by a -compiled- filter expression, whereas the optimised version has reduced this to a boolean expression in terms of the conditions. The conditions themselves are evaluated within the bodies of methods, as illustrated in section 5.3. Apart from this both versions use the same scheduling routines and method implementations.

The results of the benchmarks are shown in the following table. The tests were written in the Smalltalk-80 system and executed on a 386, 25 MHz PC. The table shows the average executions times -in milliseconds- of 25 runs, and gives the relative improvement of the optimisation.

n:	1			10			100		
b:	opt.	unopt.	Δ%	opt.	unopt.	Δ%	opt.	unopt.	Δ%
1	64.8	65.2	0.7	514.2	538.4	4.7	5034.9	5343.0	6.1
3	64.1	64.5	0.7	509.6	542.7	6.5	5071.7	5393.0	6.3
5	63.3	64.3	1.6	509.7	533.6	4.7	5068.8	5383.6	6.2
10	63.5	64.4	1.5	500.4	521.7	4.3	5068.8	5345.8	5.5
100	63.4	64.4	1.5	501.9	518.7	3.3	4859.8	5047.4	3.9

The most important result from this benchmark is that the optimisation technique gives an improvement in all the test-cases. Although the performance improvement is not a very significant percentage of the total execution time, it must be noted that only the evaluation of conditions has been optimised, whereas we measured the total execution time of the test application, including for example the application code and the task switching overhead.

If we consider the effect of the buffer size on the results, it appears that a larger buffer size in general slightly reduces the total execution time. An increase in the number of messages shows a slightly reduced execution time per message (i.e. the total execution time divided by the number of messages).

The reduction in execution time of the optimised version varies significantly with the number of messages and with the size of the buffer. Probably three aspects determine the relative amount of performance improvement:

---

1 This requires an array of *b*+1 elements.

- An increase in the number of messages sent to the buffer decreases the overhead of initialisation and task (thread) management. If the overhead is small, the optimisation of condition evaluations has more influence on the overall execution time. This increases the improvement of the optimised version.
- If the buffer size is very small a lot of messages are blocked when first received by the buffer, and must wait until other messages have been executed. This increases the amount of overhead involved in task management, and thus leads to a reduced optimisation percentage.
- On the other hand, the optimised version performs relatively better in the case of many filter (re-)evaluations. Therefore, when the size of the buffer is small related to the number of messages that are sent to it[2], a lot of messages will be blocked in the queue, which causes more filter evaluations.

Although these benchmarks indicate that the proposed optimisations are worthwhile (in fact they are indispensable for objects with intra-object concurrency), they cannot be used for making claims about the overhead involved in the synchronisation scheme. This is because the over-all execution time of the application has been measured, instead of the condition evaluation overhead only. However, the over-all execution time gives a more realistic impression of the effect of the optimisation, and in fact it is not feasible (in the system used) to accurately measure the overhead of the condition evaluation only.

Further Research

The implementation aspects of the composition filters model form a continuing research activity. Current work includes a Sina compiler that is written in, and generates code for, Smalltalk-80 [Koopmans 94]. This system primarily aims at providing an environment for experimenting with, and evaluating new composition filters concepts. The Smalltalk environment encourages design iteration and experimentation, and offers a number of facilities for implementing language concepts through its class library. For more efficient and practical application of composition filters, future research will address both a compiler that generates C or C++ code, and the integration of C++ with composition filters.

Another current research activity [Rein 94] addresses the implementation of Sina on parallel and distributed architectures. We are in particular interested in developing an object-oriented framework that (1) supports fully automatic code generation for a wide range of architectures, and (2) can be specialised and tailored with object-oriented techniques to optimise and fine-tune this code generation algorithm for particular architectures.

The framework consists of three high-level components: (a) architectural layer, (b) the technique layer and (c) the cost layer. The architectural layer specifies both the source and the target architecture. The source architecture is the *virtual architecture* that -in this case- Sina offers to the programmer. The target architecture is the *virtual architecture* that is offered by the operating system. This generally corresponds to the physical architecture of the target system. Architectures are defined by a number of *orthogonal* components.

---

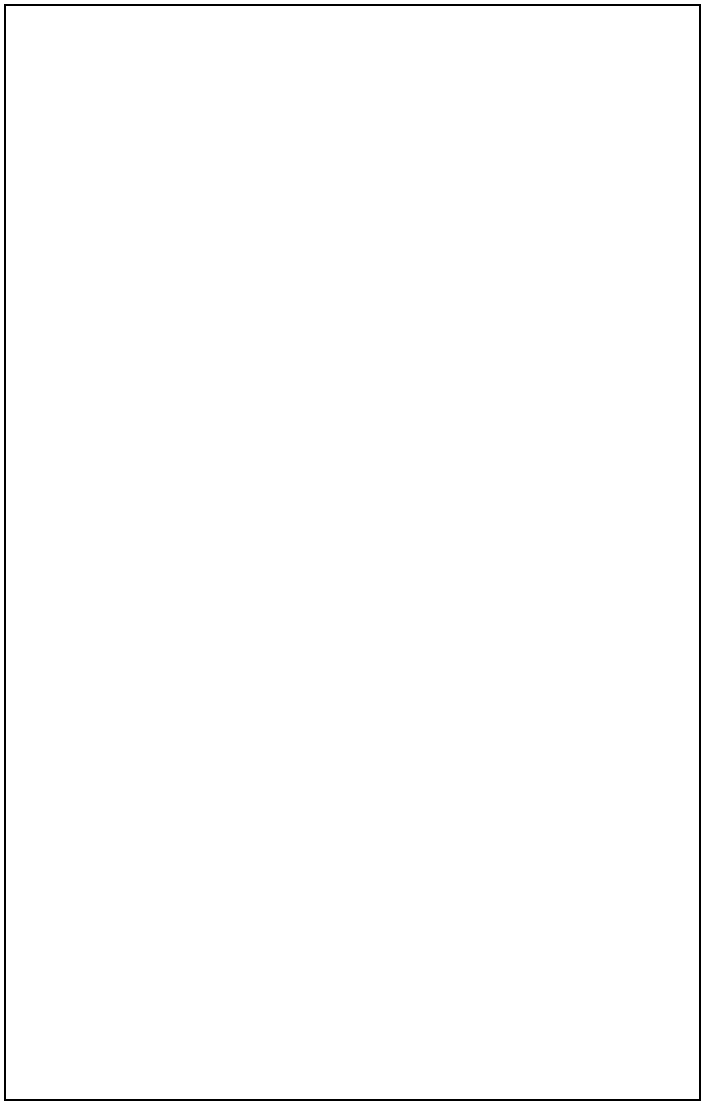
[2] Note that the amount of messages sent to the buffer is: 2*n* put messages, 2*n* get messages, and *n* peek messages, where *n* is the number specified in the header of the table.

The technique layer compares the target architecture with the source architecture and collects a number of techniques that can bridge the gap between these. For example, the shared address space assumed by the Sina language can be implemented by providing techniques for implementing message-sends to remote objects. Often there will be several alternative (sets of) techniques available; then a choice is made based on the costs of the alternatives. For determining costs, the cost layer is used. Various strategies, cq. algorithms, can be imagined for determining the optimal combination of techniques.

The cost layer models and computes the costs involved in applying specific techniques. These costs depend on both the target architecture and the characteristics of the application. The cost layer estimates the order of the number of times that a piece of code is executed. This is used to make an estimation of the total execution costs, based on the costs involved in executing a technique on the target architecture.

With this approach applications can become fully portable and architecture-independent. However, it is in general extremely difficult in such a generic approach to fully exploit the characteristics of a particular architecture *or* application. By designing the generator as an object-oriented framework, a number of hooks are available for tailoring the generator; additional architectural components can be defined, new techniques can be added, the computation of costs can be fine-tuned, and different optimisation algorithms and heuristics can be defined. With such specialisations the generator can be tailored to function optimal for certain circumstances without losing its general applicability.

Chapter 6



Conclusions

6. Conclusions

This chapter gives a brief overview of the work that has been presented in this thesis, the contributions that have been made and the perspectives for further research.

6.1 Overview and Contributions

We follow the structure of this thesis, and discuss the material in chapters 3, 4 and 5 respectively. The contents of each chapter are summarised and the contributions and points of novelty are described.

Wait Filters

Chapter 3 makes two main contributions: the first contribution is the analysis of the origins of inheritance anomalies, the second is the synchronisation scheme we propose for the composition-filters model.

In section 3.2 a generic framework for synchronisation schemes is defined. Most synchronisation schemes that perform synchronisation of received messages at the interface of the object with object-level synchronisation fit within the framework. The origins of inheritance anomalies are then explained using this framework. A classification of the origins of inheritance anomalies is given. These are used to derive a number of requirements on synchronisation schemes to avoid the inheritance anomalies.

The synchronisation scheme we propose for the composition-filters model is based on a new type of filter, the wait filter. This filter is used to specify synchronisation constraints on messages and is fully integrated within the framework offered by the composition-filters model. Concurrency is created through the transparent mechanism of non-blocking RPC message passing semantics. We show that the model can solve a range of synchronisation problems, can avoid all categories of inheritance anomalies at least to a certain extent, and adheres to all the criteria for concurrent object-oriented programming languages that were defined in section 3.1.

We emphasise that our proposal combines a powerful synchronisation scheme that has good reusability and extensibility properties with a smooth integration within the composition-filters model.

The Analysis and Design Method

The method that is described in chapter 4 is intended to support the analysis and design of concurrent objects and the invention, derivation and specification of synchronisation constraints for concurrent objects.

The method features a novel graphical notation for expressing synchronisation constraints. The notation is related to the well-known paradigm of state-transition diagrams. This provides an intuitive presentation of the life cycle of an object and the effect of the object state on the synchronisation of messages. The graphical notation has well-defined and precise semantics, which are expressible in terms of the composition filters computation model. The notation is composable, which allows synchronisation specifications to be split in a number of independent diagrams. These diagrams can be freely combined and reused

independently. The implementation gap between the graphical notation and the implementation model is bridged with a translation algorithm. This is important to support incremental development.

An object-oriented analysis and design method is introduced with active support for synchronisation aspects. This includes the analysis of synchronisation issues, the design of synchronisation strategies, the derivation of synchronisation constraints and the detection of consistency problems. An important aspect is that the method supports the invention of new, tailored synchronisation specifications, rather than being a discussion of mostly well-known, pre-defined synchronisation problems and their solutions. The method is integrated within the object-oriented paradigm, and it promotes and supports the construction of reusable and extensible objects.

Implementation Aspects

In chapter 5 we discussed a number of issues involved in the implementation of wait filters. In particular we focused on the optimisation of the evaluation of wait filters. This has been achieved in two steps; the first step is concerned with reasoning about wait filters. The second step deals with the evaluation of conditions. It attempts to determine as precisely as possible *when* a condition needs to be re-evaluated, and to *minimise* the amount of evaluation code through caching of partial results.

For reasoning about wait filters a translational semantics is defined that maps a set of wait filters to a function which expresses the synchronisation constraints of a message as a boolean expression in terms of condition values. The resulting mathematical specification of synchronisation constraints can be used for realisation or optimisation purposes, e.g. for constructing method guards, or for simplifying the synchronisation expressions. Another application is that the equivalence of synchronisation constraints can be determined. The key issue is that we port the synchronisation specifications to a well-known mathematical domain (i.e. that of boolean expressions) that better supports the manipulation and reasoning about the synchronisation constraints.

The optimisation of condition evaluations is achieved through a technique for reducing the amount of overhead involved in the evaluation of conditions. In particular if conditions and the objects they are defined for are static and self-sufficient, minimised condition re-evaluation code will be inserted only at the points where state changes are made that affect the conditions.

Important properties are that the technique is applicable for implementing state abstractions in objects that support intra-object concurrency, that the approach is suitable both for global and for incremental optimisation and that it can cope with dynamic binding. Global optimisation can in many cases statically determine the exact nature of arguments and dynamically bound pseudo-variables, which allows for better optimisations. Intermediate forms, for example when only the source code of application classes, but not of libraries is available are possible as well. However, if less information is available, more -probably redundant- condition evaluations are required.

Conclusion

The research that has been presented in this thesis demonstrates that the combination of concurrency and synchronisation with object-oriented programming is feasible. It describes the potential problems involved in this combination and proposes one particular solution to these problems, based on composition-filters technology. By providing support for the analysis and design aspects and by addressing the implementation aspects that are involved, it is attempted to ensure the practical applicability of the proposed solution.

This thesis can aid software engineers that develop concurrent object-oriented applications in several ways: first of all, the potential problems and pitfalls in reusing and extending concurrent objects are made explicit and analysed. Secondly, a technique is presented to overcome these problems, and the involved implementation aspects are addressed. Thirdly, the analysis and design of synchronisation constraints and concurrent objects is addressed, supported by an intuitive notation with a straightforward realisation.

6.2 Further Work

A wide range of interesting research topics lies still ahead of the work that has been presented in this thesis. We will briefly discuss a number of these topics in three categories; the first category is the application and extension of the current research. The second category addresses formal aspects. The third category discusses some extensions and improvements to the composition filters computation model. In section 4.6.3 further work related to the analysis and design method has been described, and in section 5.5 additional research with respect to the implementation aspects was outlined.

Application and Extension of the Current Work

The work that has been presented in this thesis has a number of natural extensions; this is partly concerned with the application of the introduced techniques, and partly with a further elaboration of the presented material. We describe a few of the issues:

- One of the important goals of object-oriented programming is the development of *application frameworks* [Johnson 88a]. An application framework offers generic solutions for a specific application domain which can be effectively reused and tailored to specific needs. An application framework forms the core of a library of reusable components.

  The development of a library with reusable components for a wide range of concurrency and synchronisation constructs is an excellent test case for verifying the applicability and reusability properties of the proposed mechanisms. Such a library may include various synchronisation techniques (cf. [Maekawa 80]) and synchronisation strategies (such as variations to reader-writer synchronisation and message passing semantics). Effectively reusable components are of prime concern for a software development process that is based on *software composition* [Nierstrasz 93a], rather than programming.

- For a broad application of the research results, it would be relevant to investigate to what extent the composition-filters approach to concurrency and synchronisation can be embedded in a programming language with pure inheritance. The composition-filters approach to achieve inheritance through object composition has particular advantages, but also some drawbacks. In particular, the conventional inheritance mechanism is more

efficient (due to reduced flexibility), and has more practical impact. At this point it is not clear yet how much functionality is to be sacrificed when making this step, except for losing some flexibility.

- An interesting topic is the integration of the method presented here within the so-called *hermeneutic* approach to software development. Hermeneutics provide a framework to explicitly model the various design decisions, hints and alternatives, and support these in a computer-aided software engineering environment. An important property of hermeneutics is that all development information is kept and used to actively support iteration and maintenance. More information on hermeneutics can be found in [Aksit 94c], [Koehorst 94] and [Algra 94].

- Yet another topic for future research is the exploitation of the state composition diagrams: in this chapter these are used only to express synchronisation constraints. However, the diagrams actually are a visual representation of filter specifications. This suggests that the same notation, or a closely related one, can be used to visualise the specification of arbitrary filter types, such as dispatch and error filters. It is not clear at this point to what extent this is possible, and whether such a notation will be effective in other domains at all.

Formal Specification and Verification

One important topic that has not been addressed in this thesis is the formal semantics of a concurrent composition-filters application. In particular, in section 2.6, we did specify the semantics of sending a message to an object. But this is not a fully formal specification, it is only concerned with the reply value that the message would return, and it is applicable in a sequential environment only. The formal aspects involved in describing the behaviour of a dynamic system of interacting concurrent objects are beyond the scope of this thesis.

- One of the complications that concurrency introduces in -formally- describing the behaviour of objects is that it requires a global perspective of an application system, since the reaction of an object to a particular message may depend on the other -concurrent- activities in the system. The timing (i.e. relative order) of the other activities in the system can be relevant for the eventual result (or even successful termination) of a message invocation. The flexibility of the composition-filters model such as dynamic binding, dynamic inheritance and run-time object creation makes it even more complicated to make statements about the behaviour of a system.

  The area of formal semantics for concurrent object-oriented systems is still an area of active research (see for instance [America 92], [Papathomas 92], [Meseguer 93a], and [Nierstrasz 93a]).

- A typing system is an approach to formally describing the properties of objects that is -in current practical systems- restricted to a few specific object properties. One of these is the signature of an object; the set of messages that are supported by the object. The mechanism for type checking Sina that has been described in section 2.5 is based on the signatures of objects. The type checking system then tries to determine whether a

specific configuration of objects is safe, which in this particular case means that every message that is sent to an object is recognised and supported by that object.

The signature of a composition-filters object depends on its filters, and is mainly determined by the Dispatch filter. By evaluating the filter specifications, static assumptions can be made about the messages that are supported, i.e. the signature of the object.

However, a typing system may include other object properties than the object signature as well. For instance, in [Aksit 89] the global objects that is referred to by an object are taken into consideration in the type-checking algorithm. In [Nierstrasz 93b] a proposal is done for typing objects based on the acceptance of messages. The notion of a *regular type* includes information about which messages are acceptable in which -abstract- object states. Type substitution is based on the notion that a subtype should accept requests at least when the supertype accepts them[1].

A similar consideration is to view filter specifications as a type specification; each filter describes a particular aspect of the externally visible object properties. Just like this information is used for deriving the -message- signature of the object, it can be used for deriving other object properties. An extended type checking algorithm may take this additional information into account, and apply it for avoiding system inconsistencies. Further research is required to investigate this and determine its applicability.

- An additional aspect is the verification of -synchronisation- specifications for objects. Applying the steps in the method proposed in chapter 4 results in a number of separate object specifications with individual synchronisation constraints. The interplay between the various objects in a -dynamic- configuration of objects determines the eventual results that the application will produce. One particular interest would be to verify to what extent such an application meets a formal specification of the overall application behaviour. As this requires a semantics of a dynamic system of interacting objects, this appears to be a very complicated research topic, unless the description of the overall application behaviour can be directly mapped to the object descriptions.

Improvements and Extensions to Composition Filters

The composition filters framework has offered the enabling technology for the research contributions in this thesis. In return, the research on concurrency and synchronisation has provided us with experience and feed-back on the applicability of composition filters in this domain. A number of conclusions are given here:

- Each composition filter specification of an object specifies a particular property of the object, by manipulating the messages that the object receives (or sends). All the

[1] Although this rule avoids surprises for the clients of an object, there may be an interference with specialisation through inheritance, where it is attractive to impose additional constraints on request acceptance (in fact, in our approach it is not allowed to relax the synchronisation constraints for inherited methods). We have similar objections to this view of substitutability as those mentioned in [Frølund 92].

messages that appear at a filter-set are subject to manipulation by all the filters in that set. However, it frequently occurs that one filter specification is only concerned with a restricted set of messages. For instance, synchronisation specifications through wait filters often deal with constraints on a small set of messages. Such filter specifications are then specified in the following form:

```
aFilter : Wait = { Constr1=>m1, ..., Constr2=>mi, True~>{m1, m2,...,  mi} }
```

The last filter element in this specification rules out all the messages that are not explicitly dealt with in this particular filter. However, this requires an explicit enumeration of all the messages, which is tedious and obscures the intentions of the filter. From this perspective, it is attractive to extend the filter specification syntax with an else-clause that specifies acceptance for messages that do not appear in the filter specification. An example of such an extension:

```
aFilter : Wait = { Constr1=>m1, ..., Constr2=>mi } else True;
```

The else clause in this example specifies a condition, maybe even a complete filter element would be appropriate as well.

- As the composition-filters framework is used for addressing problems in more application domains, the need for even more powerful specification techniques may arise. It is possible that the introduction of *nested filters* is needed to tackle new problems. Nesting of filters means that each filter element in the current filter specifications can be a filter by itself. The power of this is that it introduces an abstraction technique for filters; a set of filters can be grouped and named, filters can be composed themselves (!) and the stream of messages that arrives at the filter set can be split and manipulated. Although there is little doubt that nested filters increase the expressive power for composing object behaviour, it is far from clear whether this makes up for the increased complexity.

- From the perspective of reusability and extensibility, the current approach to specifying filter sets is not completely satisfactory. The definition of a filter is now combined with its instantiation. Instead, the definition and instantiation should be separated, so that different filter specifications can be instantiated without affecting other objects that reuse filter definitions. In the current situation, encapsulation is broken because the implementation of a filter is not fully separated from its declaration on the interface of the object.

- A promising direction for the development of new filter types lies in the definition of filters that allow *multiple dispatch*. This means that a single message can trigger the dispatch of multiple methods. Possible applications are in the creation of parallelism or attaching pre-actions and post-actions to -reused- messages.

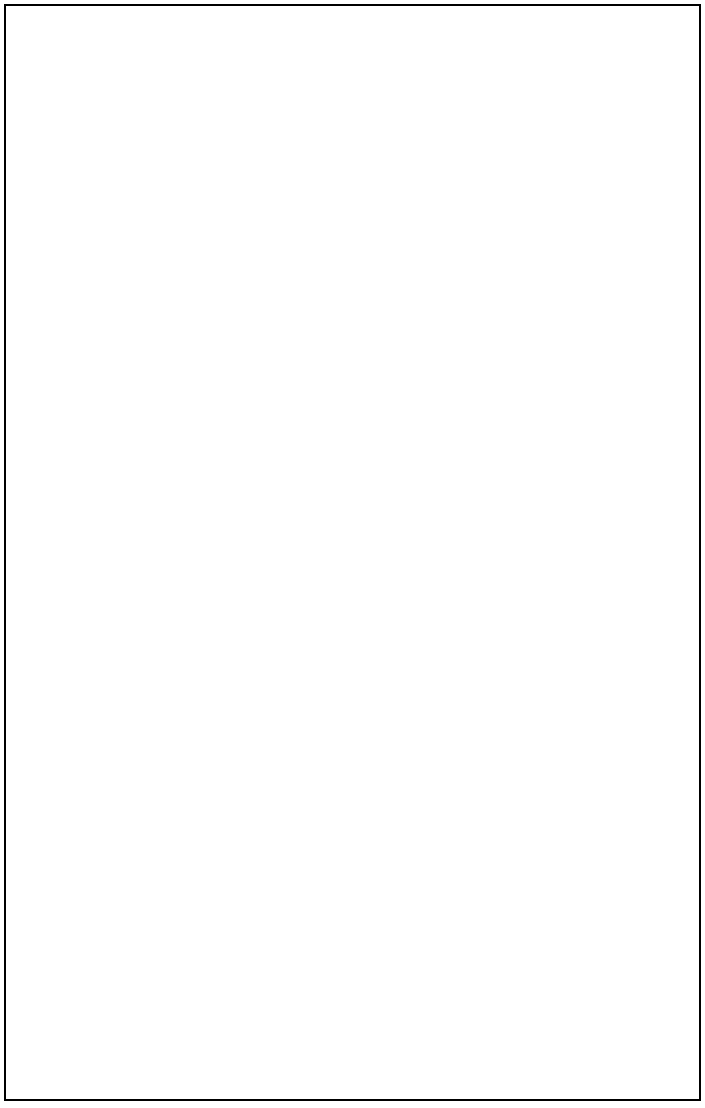
- Another issue that deserves further research efforts is a more seamless integration of atomic delegations with composition filters; instead of the dedicated syntactic construct for specifying atomic delegations, the same results can maybe be obtained by defining an appropriate filter type. A multiple, atomic dispatch seems to be feasible.

  The topic of atomic transactions requires even more attention, to address the issue of the interaction between wait filters and atomic delegations. An important aspect of atomic

delegations is that it involves synchronisation between competing objects. It should be investigated whether atomic delegations and wait filters can interfere with each other. The relation between a framework for transactions with flexible semantics [Tekinerdogan 94] and atomic delegations should be investigated as well.

From the preceding discussion it should be clear that in the areas of concurrent object-oriented software development and composition-filters technology still a lot of interesting research lies ahead of us.

Appendices



Appendix A. The Formal Notation

At a number of places in this thesis we deemed a formal presentation of definitions and specifications appropriate. We adopted a notation that is very similar to the METANOT notation [Meyer 90], which largely adheres to mathematical conventions, for these descriptions. Below the most important aspects of this notation are briefly described.

Defining Abstract Syntax

An abstract syntax is a syntax that focuses on the structure of a language, ignoring the lexicographical details. It can also be used to describe a data structure (in fact, it could be stated that it *is* a description of a data structure). An abstract syntax is described by an *abstract grammar*. The grammar consists of a number of rules, or productions, each defining the structure of a *construct*:

$Construct \stackrel{def}{=} ....$

On the right hand side, three types of productions are allowed: *aggregate*, *choice* and *list*. The aggregate defines a construct with a fixed number of components, where each component is tagged:

$Aggregate \stackrel{def}{=} tag1: Component1; tag2: Component2; ..$

An occurrence of an aggregate construct can be denoted as:

$Aggregate(tag1:comp1, tag2:comp2)$

Choice productions define a number of alternatives for a construct:

$Choice \stackrel{def}{=} tag1: Alternative1 \mid tag1: Alternative1 \mid ...$

Finally, the list productions designate a construct to consist of zero or more, or one or more, occurrences of another construct:

$List0 \stackrel{def}{=} OtherConstruct^*$
$List1 \stackrel{def}{=} OtherConstruct^+$

The denotation for an occurrence of a list production is as follows:

$< el1, el2, ... >$

With these productions, complete abstract grammars can be defined. A single component *k* of an aggregate *c* can be replaced with a new $value_{new}$:

c **except** $k=value_{new}$

Suppose $c \stackrel{def}{=} < ...., k:value_{old}, ... >$ before, than the new value of *c* is exactly the same, except that component *k* now has the value $value_{new}$.

Meaning Functions

We can define meaning functions for the constructs of an abstract grammar. These functions take two sets of arguments: one set is an occurrence of the construct, and one set is a number of additional arguments. We illustrate this with an example:

$Object : Message \mapsto Object \times Object$
$Object [\![obj]\!](mess) \stackrel{def}{=} FilterSet [\![obj.inputFilters]\!](mess, obj).$

The meaning function *Object*[[ ]]*()* takes a singe argument of type *Message* (and of course an argument of type *Object*), and delivers the cartesian product *Object* × *Object*. The meaning function itself is written with two types of brackets: Object[[obj]](mess). The double brackets "[[" and "]]" embody the argument from the abstract syntax, further arguments are provided between the round brackets. In the example the definition of the meaning function is again expressed in terms of another meaning function, *FilterSet*[[ ]]*()*.

A number of control structures are available for specifying the definition of the meaning fucntions:

Conditionals

A conditional expression can actually be considered as a function, which returns the value of one of its branches, depending on the value of a boolean expression. It is expressed in the following form:

if <*boolean expression*> **then**
  .. *some value*..
else
  .. *some value*..
end

Note that both branches are obligatory, as the expression must always return a result. Multiple cases are supported as follows:

if <*boolean expression*> **then**
  .. *some value*..
elseif <*boolean expression*> **then**
  .. *some value*..
else
  .. *some value*..
end

Choice-statement

The case statement allows to switch between a number of alternatives, depending on the *semantic domain* (or informally: *type*) of an expression. This is expressed as follows:

case <expression> **of**
  $type_1 \Rightarrow expression_1$ |
  $type_2 \Rightarrow expression_2$ |
  ...
  $type_n \Rightarrow expression_n$
end

An important property of this construct is that when <*expression*> is used in $expression_i$, it is considered to be an expression from the semantic domain $type_i$.

List Manipulation

A list construct is defined by a list production such as:

MyList $\stackrel{def}{=}$ *Element**

The denotation for such a list is as follows:

< Element(..1..), Element(..2..), ... , Element(..n..) >

Thus, an empty list is denoted as:

<>

The following operators can be applied to a list *l*:

l.EMPTY is a boolean value, which is *true* when list *l* is empty, and *false* otherwise.

l.LENGTH is a non-negative integer value, indicating the number of elements in *l*

l(*i*) is the *i*th element of list *l*.

l_1 ++ l_2 is the concatenation of lists l_1 and l_2.

l.FIRST = *l*(1), the first element of list *l*.

l.LAST = *l*(*l*.LENGTH), the last element if list *l*.

l.HEAD is the list *l* with the last element removed.

l.TAIL is the list *l* without the first element.

In order to ease iteration over lists, the following construct is available:

over *l* **apply** *f* **combine** *cop* **empty** *val0* **end**

which is equivalent to a function *combine* that is defined as follows:

combine*(* l:*Alist)* $\stackrel{def}{=}$ **if** l.*EMPTY* **then** val_0 **else** cop*(* f*(*l.*FIRST)*,* combine*(* l.*TAIL))* **end**

where

l *: ListType**

f *: ListType* $\rightarrow$ *NewType*

cop *: NewType* $\times$ *NewType'* $\rightarrow$ *NewType'*

val_0 *: Newtype*

Thus, *l* is assumed to be a sequence, *f* is a function defined on the domain of the elements of list *l*, *cop* is a combination operator that merges the result of applying *f* to an element of the list with the merged result of the rest of the list. *val0* is the value that should be returned for an empty list. The symbol # may be used to refer to the index of the current element.

Appendix B. The Sina Syntax

Below the syntax of the Sina language, *including* the syntactic sugar that is added by the pre-processor, is given. The pre-processor, among other things, allows for writing mathematical expressions in a standard format, rather than writing them as message expressions. The grammar rules are shown in alphabetic order. The rule 'SinaGrammar' is the root of the syntax.

The meta-operators are bold, and have the following definitions:

x **OPT** ≝ x **|** .
x **SEQ** ≝ [x **SEQ**] x **|** x.
x **LIST** y ≝ [x **LIST** y] y x **|** x.

The terminals always appear between quotes, non-terminals always start with a capital.

```
AndOrExpression      ≝ Expression '&&' Expression  |  Expression '||' Expression .
AtEmpNotExpression   ≝ '@' Expression  |  '&' Expression  |  '!' Expression .
AssignExpression     ≝ Expression ':=' Expression .
BinaryExpression     ≝ PlusMinExpression | MultDevExpression | AndOrExpression
                         | RelationalExpression | AssignExpression .
Block                ≝ '[' ParamDecl OPT  (Expression LIST ';' ) ']' .
BracketExpression    ≝ '(' Expression ')'.
ClassDefs            ≝ (InterfacePart ImplementPart OPT )SEQ.
ClassDescr           ≝ ClassName ( '(' Expression LIST ',' ')' ) OPT.
ClassInit            ≝ '(' ObjectDecl LIST ';'  ')' .
ClassName            ≝ Identifier .
Comment              ≝ <any text without a semicolon> ';' .
CompareExpression    ≝ Expression '=' Expression
                         | Expression '<>' Expression
                         | Expression '>' Expression
                         | Expression '<' Expression
                         | Expression '>=' Expression
                         | Expression '<=' Expression .
Condition            ≝ ConditionId ( '(' '#' ')' )OPT .
ConditionId          ≝ Identifier .
ConditionsPart       ≝ 'conditions' (
                             (ObjectName '.')OPT  ConditionId  ('(' Identifier ')')OPT  ';'
                           )SEQ OPT .
ConditionSet         ≝ Condition |  '(' Condition LIST ','  ')' .
ConstantExpr         ≝ Identifier |  Block |  Number | String .
Element              ≝ MatchPart OPT Substpart |  '<' Element LIST ','  '>' .
Expression           ≝ Object | ConstantExpr | Expression '.' MessSel | PreProcExpr .
ExternalsPart        ≝ 'externals' (ObjectDecl ';')SEQ OPT .
ExclOper             ≝ '=>' | '~>' .
Filter               ≝ '{' ( (ConditionSet ExclOper)OPT MessPattern) LIST ','  '}' .
FilterFrame          ≝ 'inputfilters' (  (
                             (FilterId ':' FilterHandler '=' Filter)
                             | (ObjectName '.')OPT FilterId
                           ) ';' ) SEQ .
FilterHandler        ≝ Identifier .
FilterId             ≝ Identifier .
```

```
IfElseExpression      def 'if' Expression
                          'then'  Expression LIST ';'
                          'else'  Expression LIST ';'  'endif' .
IfExpression          def 'if' Expression 'then'  Expression LIST ';'  'endif' .
ImplConditionsPart    def 'conditions' (
                              MethodDef  'begin' Expression 'end' ';'
                          ) SEQ OPT .
ImplementPart         def 'class' ClassName 'implementation'
                          Comment OPT
                          InstvarsPart OPT
                          ImplConditionsPart OPT
                          InitialPart OPT
                          ImplMethodsPart OPT
                          'end' ';' OPT .
ImplMethodsPart       def 'methods' ( MethodDef MethodImpl ';') SEQ OPT .
InitialPart           def 'initial' (MethodImpl ';') OPT .
InstvarsPart          def 'instvars' ( ObjectDecl ';') SEQ OPT .
InterfaceMethodDecl   def 'abstract' OPT MethodDecl .
InterfacePart         def 'class' ClassName ClassInit OPT 'input'
                          Comment OPT
                          ExternalsPart OPT
                          InternalsPart OPT
                          ConditionsPart OPT
                          MethodsPart OPT
                          FilterFrame 'end' ';' OPT .
InternalsPart         def 'internals' (ObjectDecl ';') SEQ OPT .
MainPart              def 'main' MethodImpl .
MatchPart             def '[' (MatchTarget '.')OPT MatchSelector ']' .
MatchSelector         def Identifier | '*' .
MatchTarget           def Identifier |  '*' | 'self' .
MessPattern           def Element  |  '{' Element LIST ','  '}' .
MessSel               def Identifier ( '(' Expression LIST ','  ')' ) OPT .
MethodDecl            def Identifier TypeList OPT 'returns' ClassDescr .
MethodDef             def MethodName ParamDecl OPT .
MethodImpl            def Comment OPT TempsPart OPT  'begin' Expression LIST ';'  'end'.
MethodName            def Identifier .
MethodsPart           def 'methods'  (InputMethodDecl ';') SEQ OPT .
MinusExpression       def '-' Expression .
MultDevExpression     def Expression '*' Expression  | Expression '/' Expression .
Object                def 'inner' |  'self' |  'server'  |  'sender' |  '^self' .
ObjectDecl            def ObjectName LIST ','  ':' ClassDescr .
ObjectName            def Identifier .
OtherExpression       def IfExpression | IfElseExpression
                          | WhileExpression | BracketExpression .
ParamDecl             def '('  ObjectDecl LIST ';'  ')' .
PlusMinExpression     def Expression '+' Expression  | Expression '-' Expression .
PreProcExpr           def AtEmpNotExpression | MinusExpression | ReturnExpression
                          | BinaryExpression | OtherExpression .
RelationalExpression  def Expression  '='  Expression
                          | Expression  '<>' Expression
                          | Expression  '>' Expression
                          | Expression  '<' Expression
                          | Expression  '>=' Expression
                          | Expression  '<=' Expression .
ReturnExpression      def 'return' Expression .
```

```
SinaGrammar      ≝ ClassDefs MainPart OPT  |  ClassDefs OPT MainPart .
String           ≝  '`' Astring '`' .
SubstPart        ≝  (SubstTarget '.')OPT SubstSelector .
SubstSelector    ≝ Identifier | '*' .
SubstTarget      ≝ Identifier | '*' | '#' | 'self' .
TempsPart        ≝ 'temps' (ObjectDecl ';') SEQ OPT .
TypeList         ≝ '('  ClassDescr LIST ','  ')' .
WhileExpression  ≝ 'while' Expression 'do'  Expression LIST ';'  'endwhile'.
```

Appendix C. The Interfaces of System Classes in Sina

This appendix gives the interface definitions of the most important system classes that have been used throughout this thesis. The classes that are described are Object, ObjectManager, ActiveMessage and Message. These classes are all primitive classes, which means that their implementation is hidden.

Object

Class Object defines common behaviour for all objects in the system; the pre-processor by default inserts Object as one of the superclasses of an object.

```
class Object interface
    comment this class implements a lot of the default behaviour for objects;
    conditions
        Free;        // is true when there are no active threads within the object
        Recursive;   // indicates whether the current message is a recursive message
        Protected;   // is true if the message is sent by one of its encapsulating objects
                     //   i.e. if the sender is a subclass.
    methods
        copy returns Any
            comment makes a copy of the server object;
        equals(Any) returns Boolean
            comment compares the server with the argument;
        ref returns Pointer
            comment returns a Pointer object, referring to the server;
        print(String) returns Nil
            comment displays the argument on the screen;
        error(String) returns Nil
            comment displays the argument and terminates the thread;
        stringRepr returns String
            comment returns string representation of the server object;
    inputfilters
        defSync : Wait = { Recursive=>*, Free=>* };
        defDispatch : Dispatch = { default.*, inner.* };
    outputfilters
        defSend : Send = { [*]* };
end // class Object interface
```

ObjectManager

The object manager monitors the activities within the object and the filters, and provides a set of methods that can query the status of these activities:

```
class ObjectManager interface
    comment this represents the interface that is offered by the object manager;
```

```
    methods
    // Filtering counters:
        in(Identifier) returns Integer;
            comment The number of messages received at filter <filterId>
        inFor(Identifier, Identifier) returns Integer;
            comment The number of messages <messageId> received at filter <filterId>
        out(filterId) returns Integer;
            comment The number of messages that passed filter <filterId>
        outFor(filterId, messageId) returns Integer;
            comment The number of messages <messageId> that passed filter <filterId>
    // Execution counters:
        started returns Integer;
            comment The number of started local methods.
        startedMethod(methodId) returns Integer;
            comment The number of started local methods <methodId>.
        done returns Integer;
            comment The number of terminated or aborted method executions.
        doneMethod(methodId) returns Integer;
            comment The number of finished <methodId> methods.
    // Derived counters:
        blocked(filterId) returns Integer;
            comment "The number of messages currently blocked at <filterId> ( = in(filterId)-
                    out(filterId) )."
        blockedFor(filterId, messageId) returns Integer;
            comment "The number of messages <messageId> that are currently blocked at
                    filter <filterId>"
        blockedReq returns Integer;
            comment "The sum of all currently blocked requests in all message queues of the
                    input filters.
        blockedReqFor(messageId) returns Integer;
            comment The sum of all currently blocked requests <messageId>.
        blockedInv returns Integer;
            comment "The sum of all currently blocked message invocations in all message
                    queues of the output filters"
        blockedInvFor(messageId) returns Integer;
            comment The sum of all blocked invocations <messageId>
        active returns Integer;
            comment The number of currently executing local methods ( = started-done)
        activeMethod(methodId) returns Integer;
            comment The number of currently executingmethods <methodId>
    inputfilters
        disp : Dispatch = { inner.* };
end // class ObjectManager interface
```

ActiveMessage

The pseudo-variable *message* is an instance of class ActiveMessage: it allows access to the properties of the currently executed message, but does not allow modifications. The names of most methods are self-explanatory:

```
class ActiveMessage interface
    methods
        selector returns String;
        argAt(Integer) returns Any;
        argOfAtEquals(String, Integer, Any) returns Boolean
```

```
            comment "for convenience: returns true only if the message selector is equal to
                the first parameter and the message argument indicated by the second
                parameter is equal to the value of the third parameter. Especially useful in
                conditions" ;
        sender returns Any;
        receiver returns Any
            comment this is the same as the target, or server of the message;
        self returns Any;
        inner returns Any;
        recursive returns Boolean comment "returns a boolean indicating whether this is a
            recursive message (i.e. a message to inner, self, server or sender)";
end // class ActiveMessage interface
```

Note that the pseudo-variables *self*, *server*, *sender*, and *inner* can also be obtained by accessing the *message* pseudo-variable.

Message

Meta-filters reify active messages into instances of class Message. This class allows accessing and modifying message properties, and has a few operations for manipulating messages and transforming a reified message back to an active message:

```
class Message interface
    methods
        selector returns String;
        argAt(Integer) returns Any;
        argOfAtEquals(String, Integer, Any) returns Boolean
            comment "for convenience: returns true only if the message selector is equal to
                the first parameter and the message argument indicated by the second
                parameter is equal to the value of the third parameter. Especially useful in
                conditions" ;
        sender returns Any comment returns nil when the sender is undefined;
        receiver returns Any;
        self returns Any;
        inner returns Any;
        recursive returns Boolean comment "returns a boolean indicating whether this is a
            recursive message (i.e. a message to inner, self, server or sender)";
    // all the previous messages are also supported by class ActiveMessage
        setSelector(String) returns Nil
            comment replaces the selector;
        setArgAt(Integer, Any) returns Nil
            comment set one of the message arguments;
        setReceiver(Any) returns Nil
            comment set the receiver of the message to the argument;
        resetSender returns Nil
            comment resets the sender of the message to be undefined;
        setRecursive(Boolean) returns Nil
            comment "sets the recursive property to the supplied argument";
        copy returns Message
            comment "returns a copy of the message, with an undefined value for sender"
        fire returns Nil
```

```
            comment "converts the meta-message to an active message, executed as the
                next statement by the current thread. If the sender field is still valid, the
                message will regain its previous execution context, and will not continue the
                current method. if the sender field is undefined, the current (server) object will
                become the sender, and when the message execution terminates, the current
                method will be resumed at the next operation"
        reply(Any) returns Nil
            comment "if the sender is defined, the current thread will return the supplied
                argument as the result to the sender. If sender is undefined, nothing happens"
    end // class Message interface
```

Note that we can access, but not modify, the values of pseudo-variables *self*, *inner* and *sender*. Only the value of the receiver of the message can be modified.

Samenvatting

Het toepassen van het object-georiënteerde paradigma voor de ontwikkeling van grote en complexe software systemen biedt verscheidene voordelen, waarvan verbeterde uitbreidbaarheid en herbruikbaarheid de belangrijkste zijn. Het object-georiënteerde model leent zich bovendien bij uitstek voor het modelleren van concurrente systemen. Het blijkt echter dat uitbreidbaarheid en hergebruik van concurrente applicaties verre van triviaal is. Bovendien besteden de huidige object-georiënteerde software ontwikkelingsmethoden weinig tot geen aandacht aan het analyseren en ontwerpen van synchronisatie specificaties voor concurrente objecten.

Om deze problemen aan te pakken wordt in dit proefschrift gebruik gemaakt van het raamwerk van *composition-filters*. Dit is een uitbreiding op het object-georiënteerde model. Het proefschrift bevat een analyse van de problemen met betrekking tot het hergebruiken en uitbreiden van concurrente objecten, in het bijzonder de zogenaamde *overervings anomalieën*. Mede op basis van deze analyse worden een aantal criteria geformuleerd voor effectief hergebruik en uitbreidbaarheid in object-georiënteerde programmeertalen.

In het proefschrift worden technieken geïntroduceerd voor het creëren en synchroniseren van concurrente activiteiten. Deze zijn geheel geïntegreerd met het (object-georiënteerde) composition-filters model. Belangrijke eigenschappen van het voorgestelde object-model zijn dat alle objecten zelf hun synchronisatie verzorgen, het ondersteunen van meerdere concurrente activiteiten binnen een object en een strikte scheiding van synchronisatie specificaties en methode implementaties. De toepasbaarheid en expressiviteit van het model worden gedemonstreerd, evenals de herbruikbaarheid en uitbreidbaarheid van concurrente objecten.

Tevens wordt er aandacht besteed aan de implementatie-aspecten van het voorgestelde synchronisatie mechanisme, en worden een aantal optimalisatie-technieken gepresenteerd. Er wordt getoond hoe de synchronisatie specificaties kunnen worden vertaald naar booleaanse synchronisatie-eisen op boodschappen. Dit is zowel bruikbaar voor implementatie doeleinden als voor het redeneren over synchronisatie specificaties.

Teneinde deze technieken te gebruiken voor de ontwikkeling van concurrente applicaties wordt een software-ontwikkelingsmethode geïntroduceerd welke zich voornamelijk richt op de analyse en ontwerp van synchronisatie specificaties voor concurrente objecten. De methode omvat een grafische notatie waarmee de synchronisatie van een object kan worden gedefinieerd, alsmede een aantal methode-stappen en hints om het ontwikkelen van herbruikbare synchronisatie specificaties te ondersteunen. De grafische notatie kan algoritmisch worden vertaald naar composition-filters synchronisatie specificaties.

Het in dit proefschrift gepresenteerde materiaal kan software engineers op diverse manieren ondersteunen tijdens de ontwikkeling van concurrente object-georiënteerde applicaties: potentiële problemen in de herbruikbaarheid en uitbreidbaarheid van concurrente objecten worden expliciet gemaakt en geanalyseerd. Tevens worden er technieken voorgesteld om deze problemen op te lossen. Bovendien wordt aandacht besteed aan de analyse en het ontwerp van synchronisatie specificaties, gebruik makend van een intuïtieve maar precieze notatie.

References

[Abott 83] R. Abott, Program Design by Informal English Descriptions, Communications of the ACM, Vol. 26, No. 11, November 1998

[Ackroyd 91] M. Ackroyd & D. Daum, *Graphical Notation for Object-Oriented Design and Programming*, JOOP, pp. 18-28, January 1991

[Ada 80] Honeywell, *Reference Manual for the Ada Programming Language*, Minneapolis, Minnesota, 1980

[Agesen 93] O. Agesen, J. Palsberg & M.I. Schwartzbach, *Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance*, Proceedings ECOOP '93, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 247-267

[Agha 86] G. Agha, *An Overview of Actor Languages*, ACM SIGPLAN Notices, Vol. 21, No. 10, Oct 1986, pp. 58-67

[Agha 88] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, MA., 1988

[Agha 90] G. Agha, *Concurrent Object-Oriented Programming*, Communications of the ACM, Vol. 33, No. 9, September 1990, pp. 125-141

[Aksit 88] M. Aksit & A. Tripathi, *Data Abstraction Mechanisms in Sina/ST*, Proceedings OOPSLA '88, ACM SIGPLAN Notices, Vol. 23, No. 11, November 1988, pp. 265-275

[Aksit 89] M. Aksit, *On the Design of the Object-Oriented Language Sina*, Ph.D. Dissertation, Department of Computer Science, University of Twente, The Netherlands, 1989

[Aksit 91] M. Aksit, J.W. Dijkstra & A. Tripathi, *Atomic Delegation: Object-oriented Transactions*, IEEE Software, Vol. 8, No. 2, March 1991

[Aksit 92a] M. Aksit, L. Bergmans & S. Vural, *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, ECOOP '92, LNCS 615, Springer-Verlag, 1992

[Aksit 92b] M. Aksit & L. Bergmans, *Obstacles in Object-Oriented Software Development*, Proceedings OOPSLA '92, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 341-358

[Aksit 92c] M. Aksit, K. Wakita, J. Bosch, L. Bergmans & A. Yonezawa, *Abstracting Inter-Object Communications Using Composition-Filters.*, Memoranda Informatica 92-78, University of Twente, 1992

[Aksit 94a] M. Aksit, K. Wakita, J. Bosch, L. Bergmans & A. Yonezawa, *Abstracting Object-Interactions Using Composition-Filters*, in: *Object-based Distributed Processing*, R. Guerraoui, O. Nierstrasz & M. Riveill (eds.), to be published in the Lecture Notes in Computers Science series, Springer-Verlag, 1994

[Aksit 94b] M. Aksit, J. Bosch, W. v.d. Sterren & L. Bergmans, *Real-Time Specification Inheritance Anomalies and Real-Time Filters*, to be published in the ECOOP '94 conference proceedings, 1994

[Aksit 94c] M. Aksit & H. van Oosten, *Een Framework voor Hermeneutische Objectgeorienteerde Software-Ontwikkeling: Ontwerp en Productie in één Hand* (in dutch), submitted

References

[Algra 94] E. Algra, *Process Programming and Learning Domains in Object-Oriented Hermeneutic Software Development*, MSc Thesis, University of Twente, expected June 1994

[America 87] P. America, *POOL-T: A Parallel Object-Oriented Language*, in *Object-Oriented Concurrent Programming*, eds. A. Yonezawa, M. Tokoro, The MIT Press, Cambridge, Mass. 1987, pp. 199-220

[America 90] P. America, *A Parallel Object-Oriented Language with Inheritance and Subtyping*, Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices, Vol. 25, No. 10, October 1990, pp. 161-168, October 1990

[America 92] P. America, *Formal Techniques for Parallel Object-Oriented Languages*, Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing, Springer-Verlag, 1992, pp. 119-140.

[Andrews 81] Gregory R. Andrews, *Synchronizing Resources*, ACM Transactions on Programming Languages and Systems, Vol. 3 , No. 4, 1981, pp. 405-430

[Andrews 83] Gregory R. Andrews, Fred B. Schneider, *Concepts and Notations for Concurrent Programming*, Computing Surveys, Vol. 15 , No. 1, 1983, pp. 3-43

[Atkinson 86] R.G. Atkinson, *Hurricane: An Optimizing Compiler for Smalltalk*, Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, No. 11, Nov 1986, pp. 151-158.

[Atkinson 91] C. Atkinson, S. Goldsack, A. Di Maio & R. Bayan, *Object Oriented Concurrency and Distribution in DRAGOON*, JOOP March/April 1991, pp. 11-18, 1991

[Bal 92] H.E. Bal, M.F. Kaashoek, A.S. Tanenbaum, *Orca: A Language for Parallel Programming of Distributed Systems*, IEEE Transactions on Software Engineering, Vol. SE-18, No. 3, March 1992, pp. 190-205

[Ballard 86] M.B. Ballard, D. Maier & A. Wirfs-Brock, *Quicktalk: A Smalltalk-80 Dialect for Defining Primitive Methods*, Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, No. 11, Nov 1986, pp. 140-150.

[Bear 90] S. Bear, P. Allen, D. Coleman & F. Hayes, *Graphical Specification of Object-Oriented Systems*, Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices, Vol. 25, No. 10, October 1990, pp. 28-37

[Bergmans 91] L. Bergmans, *Object-Oriented Development: Problems & Solutions*, in: ASI seminar *Object Oriented - Zin en Onzin*, (ed.) H.J. Lim, 1991

[Bergmans 92a] L. Bergmans & M. Aksit, *Reusability Problems in Object-Oriented Concurrent Programs*, ECOOP '92 Workshop on *Object-Based Concurrency & Reuse*, June 1992, Utrecht, the Netherlands

[Bergmans 92b] L. Bergmans, M. Aksit & J. Bosch, *Composition-Filters: Extended Expressiveness for OOPLs,* OOPSLA '92 Workshop on *Object-Oriented Programming Languages: The Next Generation*, October 8, 1992

[Bergmans 92c] L.M.J. Bergmans, M. Aksit, K. Wakita & A. Yonezawa, *An Object-Oriented Model for Extensible Concurrent Systems: The Composition-Based Approach*, Memoranda Informatica, 92-87, University of Twente, 1992

[Bergmans 93] L. Bergmans & M. Aksit, *Efficiently Implementing (Synchronization-) Conditions*, position paper for the OOPSLA '93 workshop '*Efficient Implementation of Concurrent Object-Oriented Programs*', October 1993

[Bergmans 94] L. Bergmans, M. Aksit, K. Wakita & A. Yonezawa, *An Object-Oriented Model for Extensible Concurrent Systems: The Composition-Filters Approach,* manuscript under revision for publication in IEEE Transactions on Parallel and Distribution Systems

[Birtwistle 73] G. Birtwistle, O.J. Dahl, B. Myhrtag & K. Nygaard, *Simula Begin*, Auerbach Press, Philadelphia, 1973

[Björnerstedt 88] A. Björnerstedt & S. Britts, *AVANCE: An Object Management System*, Proceedings OOPSLA '88, ACM SIGPLAN Notices, Vol. 23, No. 11, November 1988, pp. 206-221

[Black 86] A. Black, N. Hutchinson, E. Jul & H. Levy, *Object Structure in the Emerald System*, Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986, pp. 78-86

[Black 87] A. Black, N. Hutchinson, E. Jul, H. Levy & L. Carter, *Distribution and Abstract Types in Emerald*, IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, January 87, pp. 65-76

[Bloom 79] Toby Bloom, *Evaluating Synchronization Mechanisms*, Seventh International ACM Symposium on Operating System Principles, pp. 24-32, 1979

[Booch 90] G. Booch, *Object Oriented Design (with applications)*, Benjamin/Cummings Publishing Company, Inc., 1990

[Bos 89] J. van den Bosch & C. Laffra, *PROCOL; A Parallel Object Language with Protocols*, Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 95-102

[Breunese 92] A. Breunese, *Design and Implementation of a Mechatronic Modeling Environment Using Object-Oriented Principles*, M.Sc. Thesis, Department of Electrical Engineering, University of Twente, The Netherlands, 1992

[Brinch-Hansen 78] P. Brinch-Hansen, *Distributed Processes: A Concurrent Programming Concept*, Communications of the ACM, Vol. 21 , No. 11, 1978, pp. 934-941

[Briot 87] J-P. Briot & A. Yonezawa, *Inheritance and Synchronization in Concurrent OOP*, ECOOP '87, pp. 32-40

[Briot 89] Jean-Pierre Briot, *Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment*, Proceedings ECOOP '89, Cambridge University Press, Nottingham, July 10-14, 1989, pp. 109-129

[Buhr 92] R.J.A. Buhr & R.S. Casselman, *Architectures with Pictures*, Proceedings OOPSLA '92, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 466-483

[Campbell 74] R.H. Campbell & A.N. Habermann, *The Specification of Process Synchronization by Path Expressions*, LNCS, 16, Springer, Berlin, pp. 89-102, 1974

[Campbell 93] R.H. Campbell & N. Islam, *CHOICES: A Parallel Object-Oriented Operating System*, in *Research Directions in Concurrent Object-Oriented Programming*, (eds.) G. Agha, P. Wegner & A. Yonezawa, MIT Press, April 1993, pp. 393-451

[Caromel 90] D. Caromel, *Concurrency: An Object-Oriented Approach*, TOOLS-2, (eds.) J. Bezivin, B. Meyer & J.M. Nerson, pp. 183-197, 1990

[Caromel 93] D. Caromel, *Toward a Method of Object-Oriented Concurrent Programming*, Comm. of the ACM, Vol. 36, No. 9, September 1993, pp. 90-101

[Chambers 92] C. Chambers, *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*, PhD thesis, Stanford University, Report No. STAN-CS-92-1420

[Chandy 84] K.M. Chandy &J. Misra, *The Drinking Philosophes Problem*, ACM TOPLAS, Vol. 6, No. 4, pp.632-646, Oct. 1984

[Chen 76] P.P.S. Chen, *The Entity-Relationship Model: Toward a Unified View of Data*, ACM TODS, Vol. 1, No. 1, March 1976, pp. 9-36

[Chien 91] A. Chien, *Concurrent Aggregates: Using Multiple-Access Data Abstractions to Manage Complexity in Concurrent Programs*, ACM OOPS Messenger, Proceedings OOPSLA/ECOOP 90 Workshop on Object-Based Concurrent Systems, Vol. 2, No. 2, April 1991, pp. 31-36

[Chien 93] A. Chien, *Supporting Modularity in Highly-Parallel Programs*, in *Research Directions in Concurrent Object-Oriented Programming*, (eds.) G. Agha, P. Wegner & A. Yonezawa, MIT Press, April 1993, pp. 175-194

[Coad 91a] P. Coad & E. Yourdon, *Object-Oriented Analysis*, 2nd Edition, Yourdon Press, 1991

[Coad 91b] P. Coad & E. Yourdon, *Object-Oriented Design*, Yourdon Press, 1991

[Coleman 92] D. Coleman, F. Hayes & S. Bear, *Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design*, IEEE Transaction on Software Engineering, Vol. 18, No. 1, January 1992, pp. 9-18

[Corradi 91] A. Corradi & L. Leonardi, *PO constraints as tools to synchronize active objects*, JOOP October 91, pp. 41-53

[Coulouris 88] G.F. Coulouris & J. Dollimore, *Distributed Systems - Concepts and Design*, Addison-Wesley, 1988

[Decouchant 91] D. Decouchant, P. Le Dot, M. Riveill, C. Roisin, X. Rousset de Pina, *A Synchronization Mechanism for an Object-Oriented Distributed System*, Proceedings of the 11th IEEE Conference on Distributed Computing Systems, May 1991

[DeMichiel 87] L. G. DeMichiel, R. P. Gabriel, *The Common Lisp Object System: An Overview*, Proceedings ECOOP '87, Springer Verlag, Paris, France, June 15-17, 1987, pp. 151-170

[Deutsch 84] L.P. Deutsch & A.M. Schiffman, *Efficient Implementation of the Smalltalk-80 system*, Proceedings POPL '84, Salt Lake City, Utah, 1984.

[Dijkstra 68] E.W. Dijkstra, *Co-operating Sequential Processes*, in Programming Languages, 1968, pp. 43-112

[Ellis 90] M.A. Ellis & B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990

[Embley 92] D.E. Embley, B.D. Kurtz & S.N. Woodfield, *Object-Oriented Systems Analysis: A Model-Driven Approach*, Prentive hall, 1992

[Ferber 89] J. Ferber, *Computational Reflection in Class-Based Object-Oriented Languages*, Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 317-326

[Frølund 92] Svend Frølund, *Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages*, Proceedings ECOOP '92, Springer-Verlag, Utrecht, The Netherlands, June/July 1992, pp. 185-196

[Frølund 93] S. Frølund & G. Agha, *A Language Framework for Multi-Object Coordination*, Proceedings ECOOP '93, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 346-360

[Gangopadhyay 93] D. Gangopadhyay & S. Mitra, *ObjChart: Tangible Specification of Reactive Object Behavior*, Proceedings ECOOP '93, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 432-457

[Gerber 77] A.J. Gerber, *Process Synchronization by Counter Variables*, ACM Operating Systems Review, Vol. 11, No. 4, October 1977, pp. 6-17

[Goldberg 83] A. Goldberg & D. Robson, *Smalltalk-80: The Language and its Implemenation*, Addison-Wesley, 1983

[Grass 86] J.E. Grass & R.H. Campbell, *Mediators: A Synchronization Mechanism*, Proceedings 6th International Conference on Distributed Computing Systems, Cambridge 1986, pp. 468-477

[Greef 91] N. de Greef, *Object-Oriented system Development*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, 1991

[Haerder 83] T. Haerder & A. Reuter, *Principles of Transaction-Oriented Database Recovery*, ACM Computing Surveys, Vol. 15, No. 4, December 1983, pp. 287-317

[Hailpern 90] B. Hailpern & H. Ossher, *Extending Objects to support Multiple Interfaces and Access Control*, IEEE Transactions on Software Engineering, Vol. 16, No. 11, November 1990, pp. 1247-1257

[Harel 87] D. Harel, *Statecharts: A Visual Formalism for Complex Systems*, Sci. Comp. Progr., Vol. 8, No. 3, June 1978, pp. 231-274

[Helm 90] R. Helm, I. Holland & D. Ganghopadhyay, *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*, OOPSLA '90, pp. 169-180, 1990.

[Hoare 74] C.A.R. Hoare, *Monitors: An Operating System Structuring Concept*, Communications of the ACM, Vol. 17 , No. 10, 1974, pp. 549-557

[Hoare 78] C.A.R. Hoare, *Communicating Sequential Processes*, Communications of the ACM, 21 , No. 8, 1978, pp. 666-677

[Holland 92] I.M. Holland, *Specifying Reusable Components Using Contracts*, ECOOP '92, LNCS 615, pp. 287-308, Utrecht, June 1992.

[Hölzle 91] U. Hölzle, C. Chambers & D. Ungar, *Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches*, ECOOP '91 proceedings, LNCS 512, July, 1991

[Hopcroft 79] J.E. Hopcroft & J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979

[Ishikawa 86] Yutaka Ishikawa, Mario Tokoro, *A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation*, Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986, pp. 232-241

[Jacobson 92] I. Jacobson, M. Christerson, P. Jonsson & G. Overgaard, *Object-Oriented Software Engineering -- A Use Case Driven Approach*, Addison-Wesley/ACM Press, 1992

[Johnson 88a] R. Johnson & B. Foote, *Designing Reusable Classes*, JOOP June/July 1988, Vol. 1, No. 2, pp. 22-35

[Johnson 88b] R. Johnson, J.O. Graver & L.W. Zurawski, *TS: AN Optimizing Compiler for Smalltalk*, Proceedings OOPSLA '88, ACM SIGPLAN Notices, Vol. 23, No. 11, Nov 1988, pp. 18-26.

[Jonge 92] E. Jonge, *Object-georienteerde Analyse, Ontwerp en Implementatie van een Batchdestillatiebesturing*, M.Sc. Thesis (in dutch), Department of Chemical Engineering, University of Twente, The Netherlands, 1992

[Kafura 89] D.G. Kafura & K.H. Lee, *Inheritance in Actor Based Concurrent Object-Oriented Languages*, ECOOP '89, pp. 131-145

[Kafura 90a] D. Kafura & K.H. Lee, *ACT++: Building a Concurrent C++ with Actors*, JOOP May/June 1990, Vol. 3, No. 1, pp. 25-37

[Kafura 90b] D. Kafura, D. Washabaugh & J. Nelson, *Garbage Collection of Actors*, Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices, Vol. 25, No. 10, October 1990, pp. 126-134

[Kafura 91] D.G. Kafura & G. Lavender, *Recent Progress in Combining Actor-Based Concurrency with Object-Oriented Programming*, ACM OOPS Messenger, Proceedings OOPSLA/ECOOP 90 Workshop on Object-Based Concurrent Systems, Vol. 2, No. 2, April 1991, pp. 55-58

[Kale 94] L.V. Kale, *Report of the Workshop on Efficient Implementation of Object-Oriented Languages*, OOPSLA '93 Addendum to the Proceedings, OOPS Messenger, Vol. 5, No. 2, April 1994, pp. 111-114

[Kallstrom 88] M. Kallstrom, Shreekant S. Thakkar, *Programming Three Parallel Computers*, IEEE Software, January 1988, pp. 11-22

[Kempf 87] J. Kempf, W. Harris, R. D'Souza & A. Snyder, *Experience with CommonLoops*, Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 214-226

[Koehorst 94] H. Koehorst, *Defining Artifacts of Preparatory Work in Hermeneutic Object-Oriented Software Development*, MSc Thesis, University of Twente, 1994

[Koopmans 94] P. Koopmans, *Parsing the Sina Language to ST-80 Code*, MSc Thesis, University of Twente, expected August 1994

[Kramer 90] J. Kramer & J. Magee, *The Evolving Philosophers Problem: Dynamic Change Management*, IEEE Tranactions on Software Engineering, Vol. 16, No. 11, November 1990, pp. 1293-1306

[Laffra 92] C. Laffra, *PROCOL-A Concurrent Object Language with Protocols, Delegation, Persistence and Constraints*, PhD thesis, Erasmus University, Rotterdam, The Netherlands, 1992

[Lieberherr 89] K. Lieberherr & I. Holland, *Assuring Good Style for Object-Oriented Programs*, IEEE Software, Vol. 6, No. 9, September 89, pp. 38-48

[Lieberman 86] H. Lieberman, *Using Prototypical Objects to Implement Shared Behavior*, Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986, pp. 214-223

[Lieberman 87] H. Lieberman, *Concurrent Object-Oriented Programming in Act-1*, in *Object-Oriented Concurrent Programming*, (eds.) A. Yonezawa & M. Tokoro, MIT Press, 1987

[Liskov 77] B. Liskov, A. Snyder, R. Atkinson & C. Schaffert, *Abstraction Mechanisms in CLU*, Communications of the ACM, Vol. 20, No. 8, August 1977, pp. 564-576

[Liskov 87] B. Liskov et.al., *Argus Reference Manual*, MIT Lab. for Computer Science, No. MIT-TR-400, November 1987

[Löhr 92] K.-P. Löhr, *Concurrency Annotations*, Proceedings OOPSLA '92, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, 1992, pp. 327-340

[Löhr 93] K.-P. Löhr, Concurrency Annotations for Reusable Software, Communications of the ACM, Vol. 36, No. 9, September 1993, pp. 81-89

[Lopes 94] C.V. Lopes & K. Lieberherr, *Abstracting Process-to-Function relations in Concurrent Object-Oriented Applications*, to appear in Proceeedings ECOOP '94

[Maekawa 80] M. Maekawa, *Classification of Process Coordination Schemes in Descriptive Power*, International Journal of Computer and Information Science, Vol. 9, No. 5, 1980, pp. 43-52

[Matsuoka 90] S. Matsuoka, K. Wakita & A. Yonezawa, *Synchronization Constraints with Inheritance: What is Not Possible- So What is?*, Tokyo University, Internal Report, 1990

[Matsuoka 91] S. Matsuoka, T. Watanabe & A. Yonezawa, *Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming*, ECOOP '91, LNCS 512, pp. 213-250, Springer-Verlag, 1991

[Matsuoka 93a] S. Matsuoka & A. Yonezawa, *Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, in *Research Directions in Concurrent Object-Oriented Programming*, (eds.) G. Agha, P. Wegner & A. Yonezawa, MIT Press, April 1993, pp. 107-150

[Matsuoka 93b] S. Matsuoka, K. Taura & A. Yonezawa, *Highly Efficient and Encapsulated Reuse of Synchronization Code in Concurrent Object-Oriented Languages*, Proceedings OOPSLA '93, ACM SIGPLAN Notices, Vol. 28, No. 10, October 1993, pp. 109-126

[Meseguer 93a] J. Meseguer, *A Logical Theory of Concurrent Objects and Its Realization in the Maude Language*, in *Research Directions in Concurrent Object-Oriented Programming*, (eds.) G. Agha, P. Wegner & A. Yonezawa, MIT Press, April 1993, pp. 314-390

[Meseguer 93b] J. Meseguer, *Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming*, Proceedings ECOOP '93, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 220-246

[Meyer 86] B. Meyer, *Genericity versus Inheritance*, Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986, pp. 391-405

[Meyer 88] B. Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988

[Meyer 90] B. Meyer, *Introduction to the Theory of Programming Languages*, Prentice Hall, 1990

[Meyer 92] B. Meyer, *Eiffel: The Language*, Prentice Hall, 1992

[Micallef 88] J. Micallef, *Encapsulation, Reusability and Extensibility in Object-Oriented Programming*, JOOP April/May 1988, Vol. 1, No. 1, pp. 12-35

[Moore 56] Moore, *Gedanken-experiments on Sequential Machines*, Automata Studies, Princeton University Press, 1956

[Moss 85] J.E.B. Moss, *Nested Transactions: An Approach to Reliable Computing*, MIT Press, 1985

[Moss 87] J.B. Moss, Walter H. Kohler, *Concurrency Features for the Trellis/Owl Language*, Proceedings ECOOP '87, Springer Verlag, Paris, France, June 15-17, 1987, pp. 171-180

[Nelson 81] B.J. Nelson, *Remote Procedure Call*, Ph.D. dissertation, CMU-CS-81-119, Carnegie-Mellon University, Pittsburgh, PA, May 1981

[Neusius 91a] C. Neusius, *Synchronizing Actions*, ECOOP '91, (ed.) Pierre America, Lecture Notes in Computer Science 512, Springer-Verlag, 1991

[Neusius 91b] C. Neusius, *Adapting Synchronization Counters to the Requirements of Inheritance*, OOPS Messenger, Vol. 2, No. 4, October 1991

[Nierstrasz 87] O.M. Nierstrasz, *Active Objects in Hybrid*, Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 243-253

[Nierstrasz 91] O. Nierstrasz, *The Next 700 Concurrent Object-Oriented Languages -- Reflections on the Future of Object-Based Concurrency*, in *Object Composition*, Centre Universitaire d'Informatique, University of Geneva, June 1991, pp. 165-187

[Nierstrasz 93a] O. Nierstrasz, *Composing Active Objects*, in *Research Directions in Concurrent Object Oriented Programming*, (eds.) G. Agha, P. Wegner & A. Yonezawa, MIT Press, 1993, pp. 151-171

[Nierstrasz 93b] O. Nierstrasz, *Regular Types for Active Objects*, Proceedings OOPSLA '93, ACM SIGPLAN Notices, Vol. 28, No. 10, October 1993, pp. 1-15.

[Palsberg 91] J. Palsberg & M. Schwartzbach, *Object-Oriented Type Inference*, Proceedings OOPSLA '91, ACM SIGPLAN Notices, Vol. 26, No. 11, November 1991, pp. 146-161

[Papathomas 91] M. Papathomas & O. Nierstrasz, *Supporting Software Reuse in Concurrent Object-Oriented Languages: Exploring the Language Design Space*, Object Composition, (ed.) D. Tsichritzis, Centre Universitaire D'Informatique, 1991

[Papathomas 92] M. Papathomas, *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming*, Ph.D. thesis No. 2522, Dept. of Computer Science, University of Geneva, 1992.

[Pascoe 86] G.A. Pascoe, *Encapsulators: A New Software Paradigm in Smalltalk-80*, Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986, pp. 341-346

[Pernici 90] B. Pernici, *Objects with Roles*, Proc. of the Conference on Office Information Systems, pp. 205-215, Cambridge (Mass.), April 1990

[Reghizzi 91] S.C. Reghizzi, G.G. de Paratesi & S. Genolini. *Definition of reusable concurrent software components*, ECOOP '91, LNCS 512, pp. 148-166, Springer-Verlag, 1991

[Rein 94] R. van Rein, *An Object-Oriented Framework for Mapping Concurrent Applications to Parallel and Distributed Architectures,* MSc Thesis, University of Twente, expected July 1994

[RICOT 94] Research Initiative on Compositional Object Technology, *Preliminary Research Report*, University of Twente, 1994

[Robert 77] P. Robert & J.-P. Verjus, *Toward Autonomous Descriptions of Synchronization Modules*, Information Processing 77, North Holland, 1977, pp. 981-986

[Rumbaugh 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy & W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991

[Rumbaugh 93] J. Rumbaugh, *Controlling Code: How to implement dynamic models*, JOOP Vol. 6, No. 2, pp. 25-30, May 1993

[Sciore 89] E. Sciore, *Object Specialization*, ACM Transactions on Information Systems, Vol. 7, No. 2, April 1989, pp. 103-122

[Shibayama 91] E. Shibayama, *Reuse of Concurrent Object Descriptions*, TOOLS-3, pp. 254-266, 1991

[Shlaer 88] S. Shlaer & S.J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice Hall, 1988

[Shlaer 92] S. Shlaer & S.J. Mellor, *Object Life-Cycles: Modeling the World in States*, Prentice-Hall, 1992

[Snyder 86] A. Snyder, *Encapsulation and Inheritance in Object-Oriented Programming Languages*, Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986, pp. 38-45

[Sterren 93] W.E.P. Sterren, *Design of a Real-Time Object-Oriented Language*, M.Sc. Thesis, Dept. of Computer Science, University of Twente, The Netherlands, Februari 1993

[Stroustrup 86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986

[Tekinerdogan 94] B. Tekinerdogan, *The Design of an Object-Oriented Framework for Atomic Transactions*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, 1994

[Tokoro 94] M. Tokoro, *The Society of Objects*, OOPSLA '93 Addendum to the Proceedings, OOPS Messenger, Vol. 5, No. 2, April 1994, pp. 3-12

[Tomlinson 89a] C. Tomlinson & V. Singh, *Inheritance and Synchronization with Enabled-sets*, Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 103-112

[Tomlinson 89b] C. Tomlinson & M. Scheevel, *Concurrent Object-Oriented Programming Languages*, in *Object-Oriented Concepts, Databases, and Applications*, (eds.) W. Kim & F.H. Lochovsky, Addison-Wesley, 1989

[Tripathi 88] A. Tripathi & M. Aksit, *Communication, Scheduling and Resource Management in Sina*, JOOP, Vol. 1, No. 4, November/December 1988, pp. 24-37

[Tripathi 89] A. Tripathi, E. Berge & M. Aksit, *An Implementation of the Object-Oriented Concurrent Programming Language Sina*, Software-Practice and Experience, Vol. 19, No. 3, March 1989, pp. 235-256

[Ungar 87] D. Ungar & R. B. Smith, *Self: The Power of Simplicity*, Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 227-242

[Wakita 91] K. Wakita & A. Yonezawa, *Linguistic Supports for Development of Organizational Information Systems,* Proceedings of ACM COCS, November 1991

[Wakita 93] K. Wakita, *First Class Messages as First-Class Message Continuations*, in *Object Technologies for Advanced Software*, (eds.) S. Nishio & A. Yonezawa, Proceedings of the first JSSST International Symposium, Kanazawa, Japan, November 1993, pp.442-459

[Ward 85] P. Ward & S. Mellor, *Structured Development for Real-Time Systems: Introduction and Tools*, Yourdon Press, 1985

[Wasserman 85] A. Wasserman, *Extending State Transition Diagrams for the Specification of Human-Computer Interaction*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 8, pp. 699-713, August 1985

[Watanabe 88] Takuo Watanabe, Akinori Yonezawa, *Reflection in an Object-Oriented Concurrent Language*, Proceedings OOPSLA '88, ACM SIGPLAN Notices, Vol. 23, No. 11, November 1988, pp. 306-315

[Wegner 87] P. Wegner, *Dimensions of Object-Based Language Design*, Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 168-182

[Wegner 90] P. Wegner, *Concepts and Paradigms of Object-Oriented Programming*, OOPS Messenger, No. 1, Vol. 1, August 1990, pp. 7-87

[Wegner 91] P. Wegner, Panel on Object-based Concurrent Programming, in ACM OOPS Messenger, Proceedings OOPSLA/ECOOP 90 Workshop on Object-Based Concurrent Systems, Vol. 2, No. 2, April 1991, pp. 4-7

[Wegner 92] P. Wegner, *Design Issues for Object-Based Concurrency*, Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing, Springer-Verlag, 1992, pp. 245-256

[Whiting 90] M.A. Whiting & D.M. DeVaney (eds.), *Finding the Object*, report of the ECOOP/OOPSLA workshop "Finding the Object", October 22, 1990

[Wirfs-Brock 89] R. Wirfs-Brock & B. Wilkerson, *Object-Oriented Design: A Responsibility-Driven Approach*, Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 71-75

[Wirfs-Brock 90a] R. Wirfs-Brock, B. Wilkerson & L. Wiener, *Designing Object-Oriented Software*, Prentice-Hall, 1990

[Wirfs-Brock 90b] R. Wirfs-Brock & R.E. Johnson, *A Survey of Current Research in Object-Oriented Design*, Communications of the ACM, Vol. 33, No. 9, September 1990, pp. 104-124

[Wyatt 92] B.B. Wyatt, K. Kavi & S. Hufnagel, *Parallelism in Object-Oriented Language: A Survey*, IEEE Software, 9, 6, Nov 1992, pp. 56-66

[Yokote 86] Y. Yokote & M. Tokoro, *The Design and Implementation of ConcurrentSmalltalk*, Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986, pp. 331-340

[Yokote 87a] Y. Yokote & M. Tokoro, *Experience and Evolution of ConcurrentSmalltalk*, Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 406-415

[Yokote 87b] Y. Yokote & M. Tokoro, *Concurrent Programming in ConcurrentSmalltalk*, Object-Oriented Programming, (eds.) A. Yonezawa & M. Tokoro, MIT Press, 1987

[Yonezawa 87] A. Yonezawa et al., *Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1*, in *Object-Oriented Concurrent Programming*, (eds.) A. Yonezawa & M. Tokoro, MIT Press, pp. 55-89, 1987

[Yonezawa 90] A. Yonezawa (ed.), *ABCL: An Object-Oriented Concurrent System*, Computer Systems Series, The MIT Press, 1990

[Yourdon 89] E. Yourdon, *Modern Structured Analysis*, Prentice-Hall, 1989

[Yücesoy 92] E. Yücesoy, *Assessment of the Composition-Filters Model*, MSc. Thesis, University of Twente, November 1992