The Composition-Filters Object Model*

Lodewijk M.J. Bergmans
TRESE, University of Twente+

In this paper 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 paper will address the basic mechanisms and how they can be applied to solve various modelling problems.

1 Introduction

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 programming language Sina that is introduced in this paper 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

---

* This paper is derived from chapter 2 of "Composing Concurrent Objects" [Bergmans 94]

+ The author can be reached at the University of Twente, Dept. of Computer Science, Group SETI, Project TRESE, P.O. Box 217, 7500 AE, Enschede, The Netherlands. Or through electronic mail as: {trese, bergmans}@cs.utwente.nl

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.

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

---

3 Or similar reuse techniques such as delegation and composition.

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.

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:

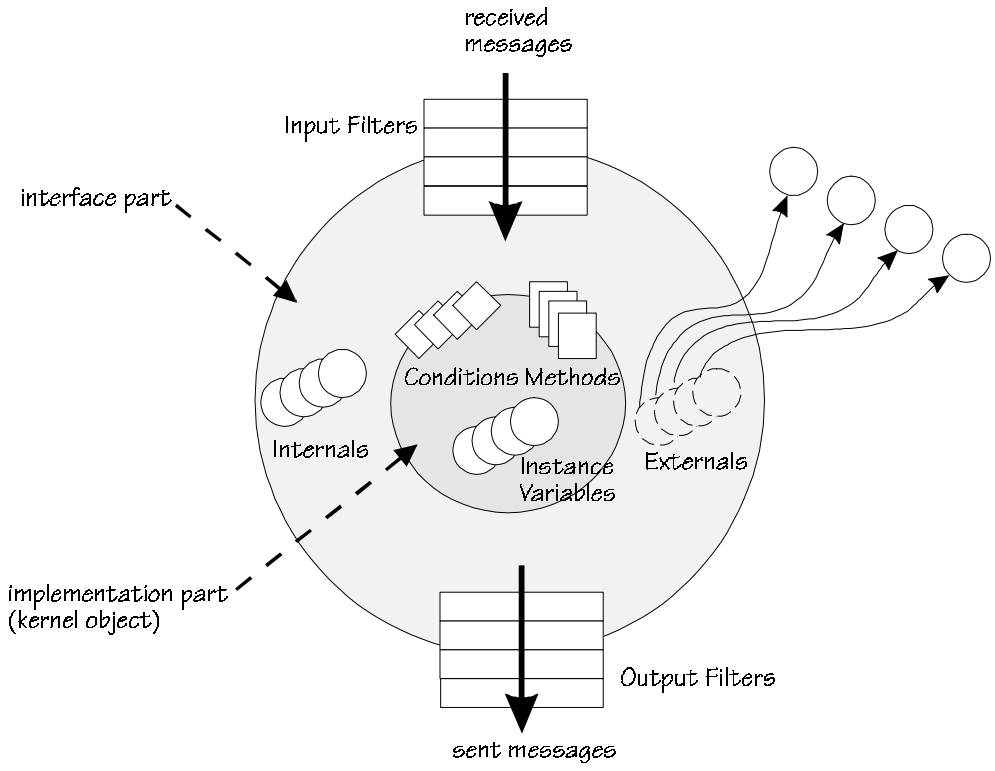

Figure 1.*1 The components of the composition-filters model.*

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 paper.

The Plan of the Paper

In the rest of this paper, 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.

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

- Associativity, or 'queries', 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], and for concurrency and synchronisation [Bergmans 94] is not discussed here.

The rest of this paper 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 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.

Figure 2.1 shows the two layers, with some additional detail of the implementation part:

The interface part receives the incoming messages, processes them in a manner that will be elaborately discussed later in this paper, 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.

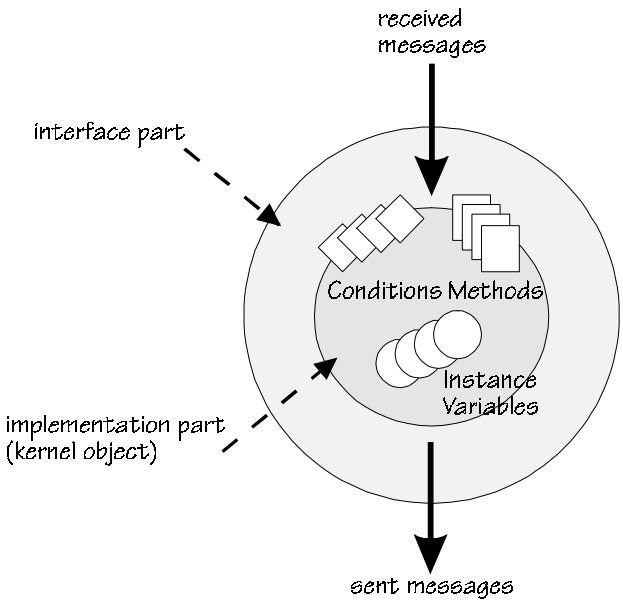

Figure 2.1 The two layers of the composition-filters object model.

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[7].

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.

---

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

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[8] 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 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:

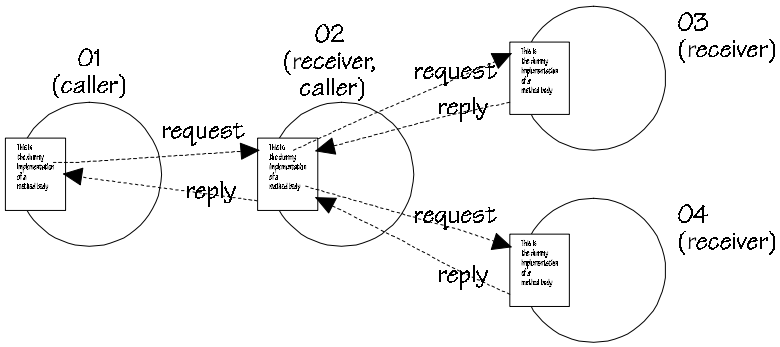

Figure 2.2 The chain of requests and replies between objects.

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;
```

---

8 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.

```
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[9]. 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[10] 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 93]. 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.

---

9 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.

10 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.

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 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[11]. 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[12]. 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.

---

[11] 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.

[12] 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.

3 Composition Filters

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:

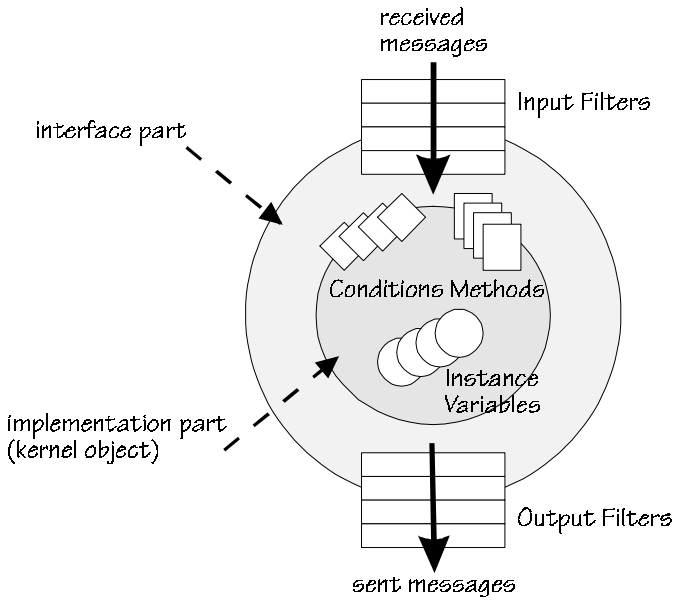

Figure 3.1 The composition-filters object model with input and output filters.

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.

3.2 The Principles of Composition Filters

We will explain the basic mechanism of composition filters with the aid of figure 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[13]. In this paper we do not bother with the issues related to concurrent activities and multiple -simultaneous- message invocations on an object.

Figure 3.2 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

---

[13] 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.

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.

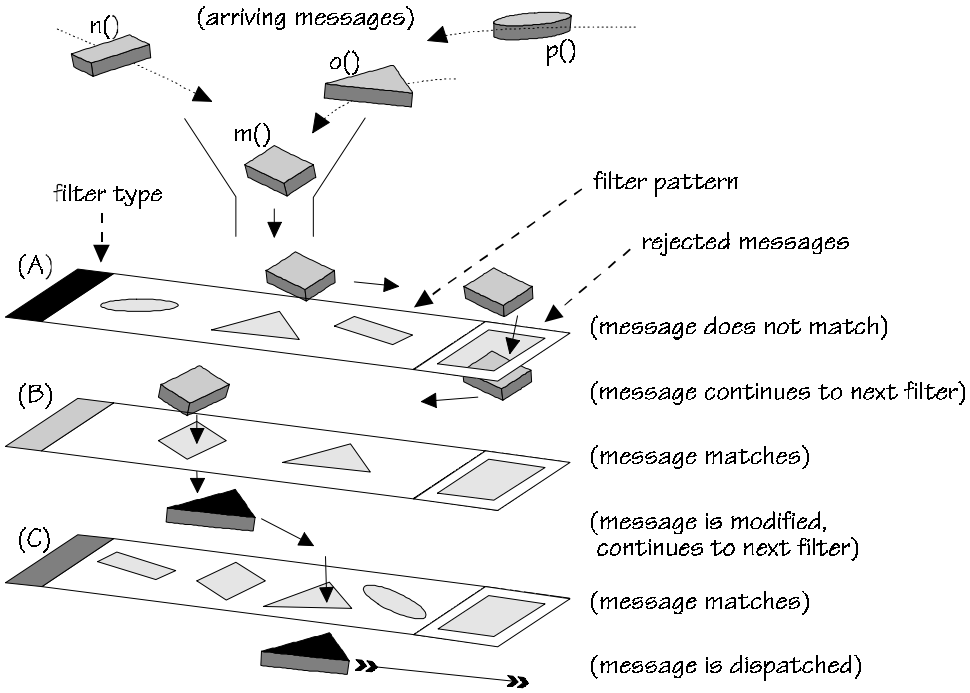

Figure 3.2 A schematic representation of the filter mechanism.

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.

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[14]. 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 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.

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.

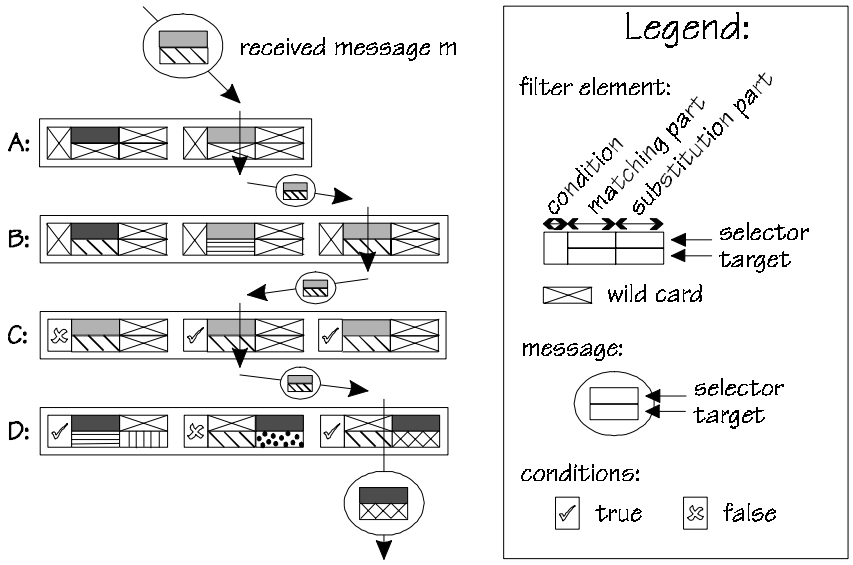

Figure 3.3 A visual representation of filter specifications

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.

---

[14] 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.

- 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 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.

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[15]. 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 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 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.

---

[15] 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 3.3: Default condition

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

Definition 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 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[16].

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 paper.

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 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 3.3, 3.4 and 3.6 to the select filter specification, to simplify the filter initialisation clause:

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

However, when considering definition 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:

---

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

Definition 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 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.

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 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*> $\stackrel{def}{=}$

```
 1  if matchingPartUndefined                                              (1)
 2  then if (substSel='*' ∨ substSel=mess.selector)                       (2)
 3          ∧ (substTar='*' ∨ mess.selector∈ sig(substTar) )              (3)
 4       then if substTar≠ '*' then mess.receiver := substTar end;        (4)
 5       else <mess, false> end                                           (5)
 6  else if (matchTar='*' ∨ matchTar=mess.receiver)                       (6)
 7          ∧ (matchSel='*' ∨ matchSel=mess.selector)                     (7)
 8       then if substTar≠ '*' then mess.receiver := substTar end;        (8)
 9            if substSel≠ '*' then mess.selector := substSel end;        (9)
10            <mess, true>                                               (10)
11       else <mess, false> end                                           (11)
12  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[17]. 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.

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.

---

[17] 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.

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.[18]
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

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 ≝ matching target
   b ≝ matching selector
   c ≝ substitution target
   d ≝ 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.

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

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.

4.1 The Components of the Interface Part

The following figure shows the components of the object model:

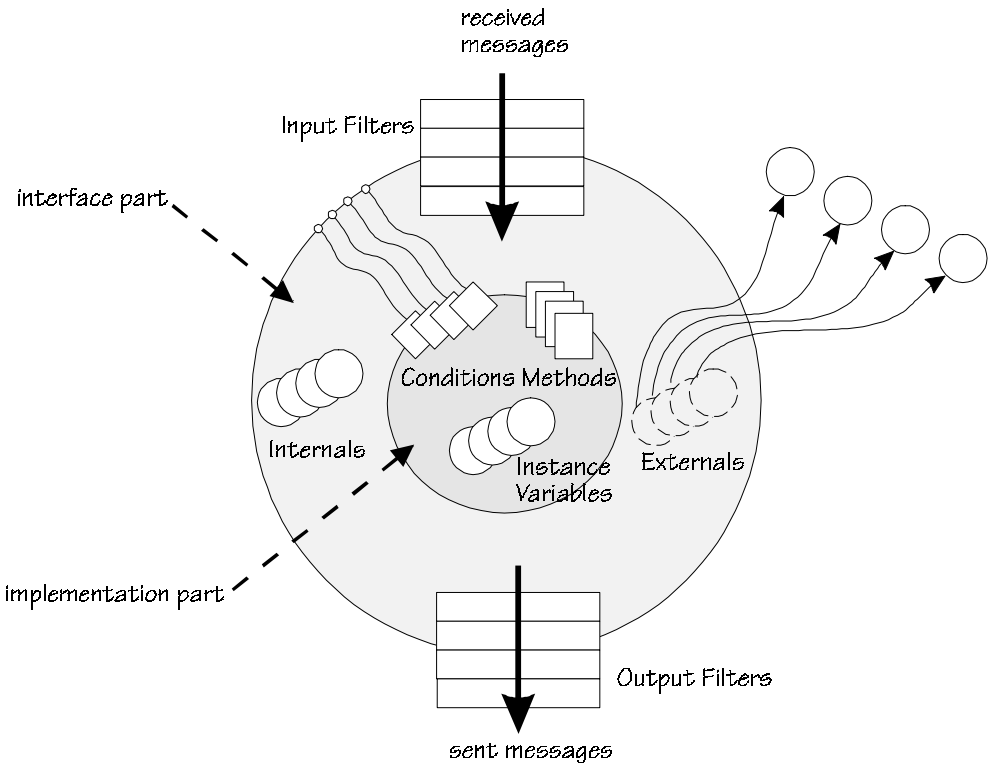

Figure 4.1 The components of the composition filters extension in the interface part.

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 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[19], such as for instance global objects. Within both the interface and the implementation part of the

---

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

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.

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
            initialisation 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 difference is the initialisation: the class parameters are initialised with values that are provided by the statement that causes the object creation[20].

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:

---

[20] 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 ✣● ▲▲, 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.

```
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";
```

In the interface part only the header, or *signature*[21], 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

---

[21] 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.

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.

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.

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:

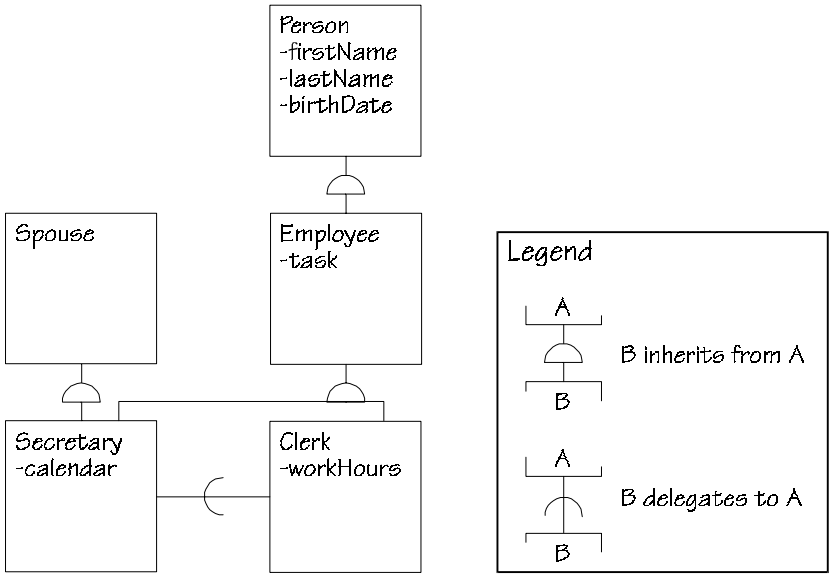

Figure 4.2 An overview of the reuse relations that are discussed in this subsection.

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[22], 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]).
>
> 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).

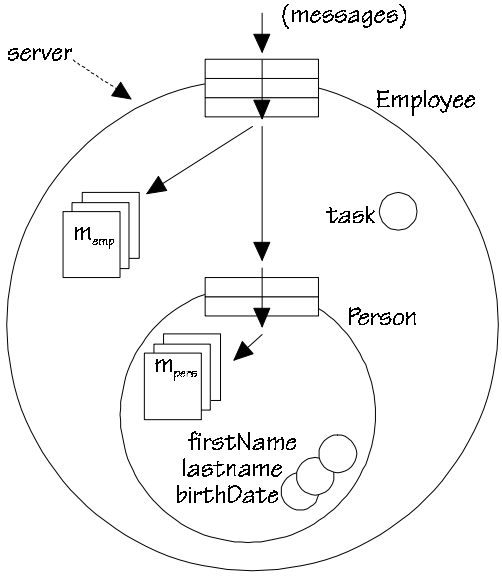

Figure 4.3 Inheritance in the composition filters model.

---

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

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.

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*[23]. This pseudo-variable always refers to the object that first received the message. By 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 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[24], then with the signature of private, and finally with that of work.

---

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

[24] 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.

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.

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:

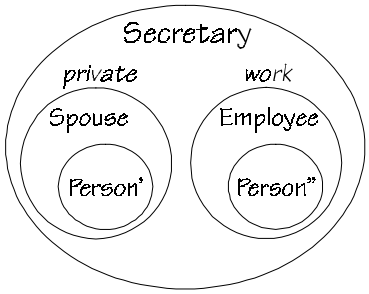

Figure 4.4 The object nesting structure of multiple inheritance with shared parents.

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 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*[25] 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 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.

---

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

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[26]. 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 88]). 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[27].

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[28].

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";
```

---

[26] 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.

[27] In object models that provide delegation, dynamic inheritance *can* be simulated, though.

[28] 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.

```
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[29].

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.

---

[29] 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.

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 [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[30].

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.

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.

---

30 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].

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[31]. 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 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].

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

---

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

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[32]. 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.

Definition 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*.

---

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

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[33].

> 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 5.1 illustrates this concept.

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.

---

33 The use of genericity is not always satisfactory in such situations, because of the following problem: consider a collection object with a ⁜⁜▼ and a ❒◆▼ operation that stores objects of a type *T* -indicated as ❋❑●●✈✳✉- that is defined through parameterisation. The problem is that, for different types *S* and *T*, ❋❑●●✈✳✉ can never be a subtype of ❋❑●●✈✳✉, even if *T* is a subtype of *S*. This is due to the contravariance between arguments (e.g. of ❒◆▼) and return types (e.g. of ⁜⁜▼).

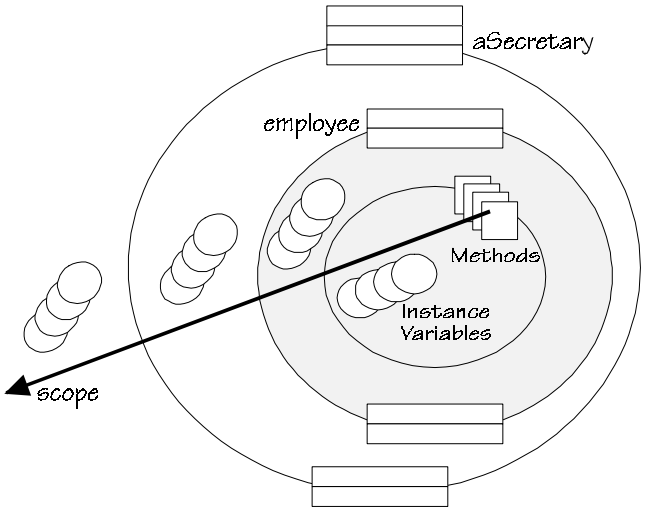

Figure 5.1 A schematic representation of the scope rules.

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:

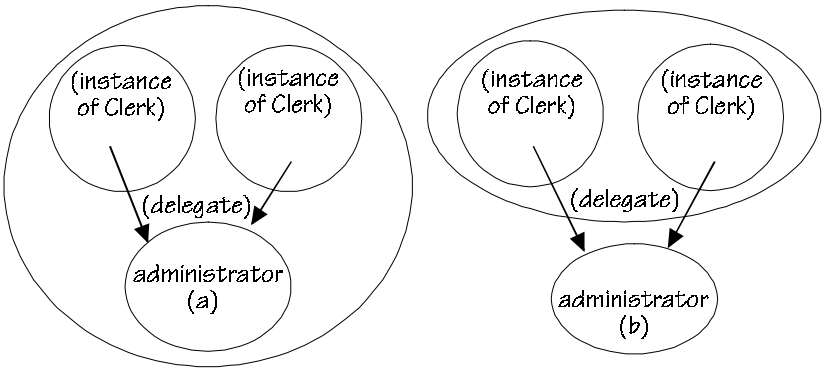

Figure 5.2 An object configuration demonstrating multiple objects with the same name.

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 an example of an execution thread shown in figure 5.3. This 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).

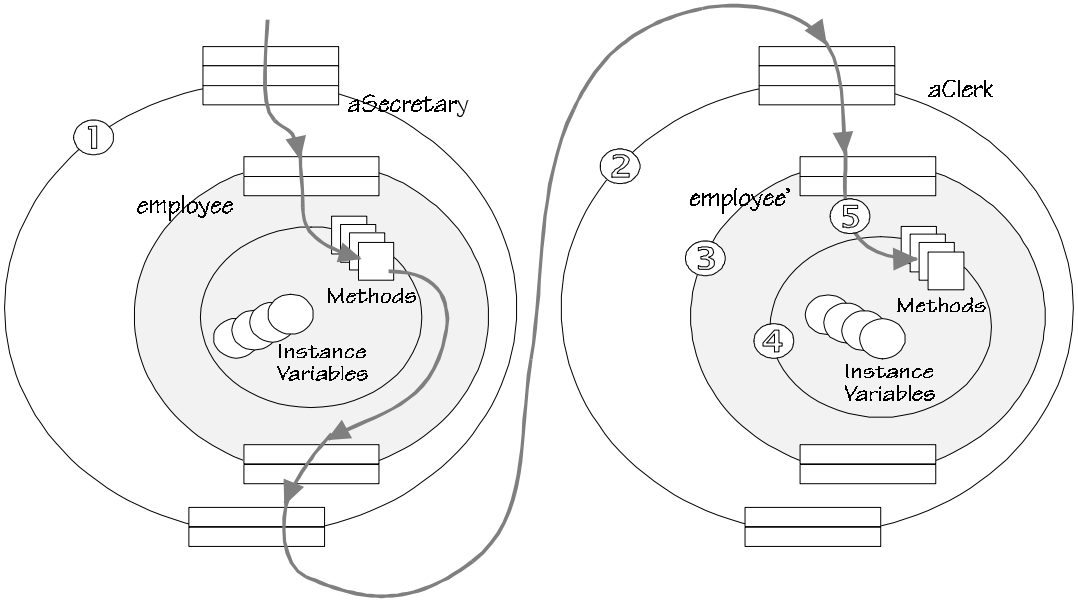

Figure 5.3 An illustration of an execution thread with the current pseudo-variables: ①=sender, ②=server, ③=self, ④=inner, ⑤=message

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[34], 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

[34] 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.

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*[35]. The Sina system contains a primitive class called Object which provides a number of standard operations. Typical examples of standard operations used in this paper 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.*;
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.

---

[35] 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*.

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[36] 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.

---

[36] 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.

Note that this requires the client of the object to send the two messages atomically[37], 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.

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.

6 Discussion

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

6.1 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.

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

---

[37] 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.

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.

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.

6.2 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 paper 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].
- ❏ Mechanisms for concurrent programming and specification of synchronisation constraints such that reusability and extensibility of active objects is supported.

References

[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

[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

[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

[Bergmans 94] L. Bergmans, *Composing Concurrent Objects*, PhD thesis, University of Twente, june 1994

[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 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

[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

[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

[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

[Ellis 90] M.A. Ellis & B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990

[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 93] S. Frølund & G. Agha, *A Language Framework for Multi-Object Coordination*, Proceedings ECOOP '93, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 346-360

[Goldberg 83] A. Goldberg & D. Robson, *Smalltalk-80: The Language and its Implemenation*, Addison-Wesley, 1983

[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,

[Holland 92] I.M. Holland, *Specifying Reusable Components Using Contracts*, ECOOP '92, LNCS 615, pp. 287-308, Utrecht, June 1992.

[Johnson 88] R. Johnson & B. Foote, *Designing Reusable Classes*, JOOP June/July 1988, Vol. 1, No. 2, pp. 22-35

[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

[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

[Laffra 92] C. Laffra, *PROCOL-A Concurrent Object Language with Protocols, Delegation, Persistence and Constraints*, PhD thesis, Erasmus University, Rotterdam, The Netherlands, 1992

[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

[Liskov 87] B. Liskov et.al., *Argus Reference Manual*, MIT Lab. for Computer Science, No. MIT-TR-400, November 1987

[Matsuoka 93] S. Matsuoka, K. Taura & A. Yonezawa, *Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages*, Proceedings OOPSLA '93, ACM SIGPLAN Notices, Vol. 28, No. 10, October 1993, pp. 109-126

[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 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

[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

[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

[RICOT 94] Research Initiative on Compositional Object Technology, *Preliminary Research Report*, University of Twente, 1994

[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

[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 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

[Wegner 90] P. Wegner, *Concepts and Paradigms of Object-Oriented Programming*, OOPS Messenger, No. 1, Vol. 1, August 1990, pp. 7-87

[Yücesoy 92] E. Yücesoy, *Assessment of the Composition-Filters Model*, MSc. Thesis, University of Twente, November 1992