An Object-Oriented Model for Extensible Concurrent Systems: The Composition-Filters Approach

Lodewijk Bergmans
Mehmet Aksit
Ken Wakita*
Faculty of Computer Science
University of Twente
Enschede, The Netherlands
&
Akinori Yonezawa
Department of Information Science
University of Tokyo
Tokyo, Japan

Abstract

Applying the object-oriented paradigm for the development of large and complex software systems offers several advantages, of which increased extensibility and reusability are the most prominent ones. The object-oriented model is also quite suitable for modeling concurrent systems. However, it appears that extensibility and reusability of concurrent applications is far from trivial. The problems that arise, the so-called inheritance anomalies are analyzed and presented in this paper. A set of requirements for extensible concurrent languages is formulated. As a solution to the identified problems, an extension to the object-oriented model is presented; composition filters. Composition filters capture messages and can express certain constraints and operations on these messages, for example buffering. In this paper we explain the composition filters approach, demonstrate its expressive power through a number of examples and show that composition filters do not suffer from the inheritance anomalies and fulfill the requirements that were established.

* Currently at the Tokyo Institute of Technology, Dept. of Information Science, 2-12-1 Oh-okayama, Meguro-ku, Tokyo, 152, Japan. E-mail: wakita@is.titech.ac.jp

1. Introduction

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

The continuous growth in size and complexity of today's computer systems increase the necessity for new software development methods and language mechanisms that support a high degree of modularity, extensibility and reusability [Meyer 88]. The object-oriented paradigm, supported by software development methods (e.g. [Rumbaugh 91], [Booch 90], [Coad 91a], [Coad 91b]) and languages (e.g. [Goldberg 83], [Stroustrup 86], [Meyer 88]), lends itself well to these objectives. Object-oriented programs consist of co-operating modular entities called *objects*. The concept of *inheritance* is the basic extensibility and reusability feature of object-oriented languages. Inheritance is analogous to genetic inheritance where features are inherited from parents by their children. This appears in object-oriented languages as a specification of which features of 'parent' objects are to be adopted by 'children' objects.

In addition, it was claimed that object-oriented language constructs are also suitable for expressing concurrency and synchronization since more than one object can be active concurrently [Yonezawa 87]. However, most concurrent object-oriented languages fail in combining their concurrency and synchronization mechanisms with the inheritance structure. This failure is referred to as the *inheritance anomaly*. Ideally, development methods and language constructs should be able to express concurrency and synchronization while maintaining a high degree of modularity, extensibility and reusability.

The purpose of this paper is to introduce a new language model based on the concept of *object-composition filters*. A composition-filter is a *modular* extension to the conventional object model. It is able to express concurrency and synchronization mechanisms integrated with the concept of inheritance. This eliminates problems related to the inheritance anomalies and allows programmers to develop concurrent systems with a high degree of extensibility and reusability.

This paper is organized as follows. Section 2 describes the background and related work in the area of object-oriented concurrent programming and synchronization. Inheritance anomalies and requirements of concurrent object-oriented systems are studied in section 3. Section 4 presents the mechanisms to support extensible concurrency and synchronization using composition-filters. Section 5 contains various examples to demonstrate the capabilities of the proposed mechanisms to solve some well-known synchronization problems. We have chosen examples in four different categories: process communication and coordination, implementation of functions, scheduling, and resource management. Section 6 evaluates the

composition-filters model with respect to the requirements and example problems presented in sections 3 and 5, respectively. The last section gives conclusions.

2. Background and Related Work

2.1. Object-Oriented Language Models

A language is defined as being an *object-oriented language* if it supports *objects*, *classes* and *inheritance* [Wegner 87].

Objects are autonomous entities that respond to *messages*. A message is a request for an object to carry out one of the object's *operations*[1]. Operations of an object are a set of meaningful functions provided by, and applicable to that object. Each object provides *encapsulation* so that the implementation of operations, and internal data structures of an object are hidden from its users. In a pure object-oriented model, implementation of operations will also consists of message sends. Internal data structures are defined by *local variables*[2] within an object. The local variables of an object are only accessible through its operations. Encapsulation is useful because it hides the unnecessary detail, and since the external interface of an object is independent of the object's actual realization, the re-implementation of an object will have no effect on the other objects in the system.

A *class* abstracts and groups objects according to their common behavior. These individual objects, also called *instances*, can only maintain specific properties in the form of unique values of their local variables. Instance objects can be created from a class by requesting a so-called *new* operation.

Inheritance is a structural organization of classes, whereby a class may inherit operations and/or local variables from its *superclasses*, or may have its operations and/or local variables inherited by its *subclasses*. This structural inheritance relation provides for the organization of classes so that they can be systematically reused. Classes may also inherit from multiple parents, providing additional reusability.

As an alternative to inheritance, a technique called *delegation* has been proposed [Lieberman 86]. Delegation is a mechanism that allows objects to delegate the requests of its users to one or more designated objects. Delegation is orthogonal to the class concept, and often adopted by classless languages. Basically, delegation and inheritance mechanisms have similar characteristics in that they both can implement modular software structures that share operations. The designers of delegation-based languages claim that delegation is more powerful than inheritance because it can support dynamic evolution of systems and because delegations can be configured at run-time. With delegation, the delegated object is a part of the *extended identity* of the delegating object.

---

1 In some object-oriented languages, operations are called methods.

2 Local variables are also called instance variables.

All operations defined locally by or made available to an object through delegation or inheritance are said to be within the *signature* of that object.

2.2. Concurrency in Object-Oriented Models

2.2.1. Concepts

Traditionally, synchronizing access to shared resources has been classified as either *shared-memory* or *message-based* synchronization [Andrews 83].

In shared memory systems, solutions to various synchronization problems can be constructed using mechanisms such as *conditional critical regions* [Hoare 72], *event counters* [Reed 79], *modules* [Wirth 77], *monitors* ([Hoare 74] and [Brinch-Hansen 75]), *path expressions* [Campbell 74], and *semaphores* [Dijkstra 68].

Message-based synchronization is not restricted only to the object-oriented model but also used for synchronizing concurrent processes. Some examples are languages such as Communicating Sequential Processes (CSP) [Hoare 78], Distributed Processes (DP) [Brinch-Hansen 78] and Synchronizing Resources (SR) [Andrews 81].

The concept of encapsulating a shared resource as an active module which reacts to request messages from its clients forms the basis of *object-based concurrent programming*. The policies for the synchronization of concurrent operations invoked on this module are determined by the shared resource object. Several concurrent and distributed programming languages, such as ABCL/1 ([Yonezawa 86], [Yonezawa 90]), Act-1 [Lieberman 87], Ada [Ada 80], Argus [Liskov 87], Pool-T [America 87] and PROCOL [Bos 89], and distributed operating systems, such as Eden [Almes 85], Emerald [Black 86], ISIS [Birman 85], Nexus [Tripathi 88b] are based on this kind of approach. Act-1 is one of the so-called actor-based languages [Agha 88], which are intended primarily for constructing concurrent programs.

Object-oriented concurrent programming extends the object-based model by adding inheritance-like mechanisms. Typical examples of concurrent object-oriented languages are Act++ ([Kafura 89], [Kafura 90]), ConcurrentSmalltalk ([Yokote 87], [Okamura 91]), CEiffel [Loehr 92], Dragoon [Atkinson 91], Eiffel// [Caromel 89], Guide [Decouchant 89], Hybrid [Nierstrasz 87], Orient/84 [Tokoro 86], POOL-I [America 90], and Rosette [Tomlinson 89]. Choices [Campbell 89] is an example of an object-oriented operating system. The object-based concurrency and synchronization models, as presented in this section, are also valid for object-oriented models. Inheritance structure becomes important when the inheritance anomalies are studied in section 3.

There are three approaches to incorporate concurrency within an object-based language: concurrent executions are independent from objects, objects are *active entities* and responsible for synchronizing concurrent executions, and, thirdly, a combination of these two approaches.

The first approach is exemplified by Emerald, Smalltalk [Goldberg 83], and Trellis/Owl [Schaffert 86] where concurrent executions are created by activating *processes*. Due to the orthogonality between processes and objects, this way of creating concurrency is considered not to be *pure* object-based concurrency. For synchronizing concurrent executions, Emerald,

Smalltalk, and Trellis/Owl adopt monitors [Hoare 74], semaphores [Dijkstra 68] and *lock-blocks* [Moss 85], respectively.

In the second approach, objects are autonomous and concurrently active entities communicating with each other by sending messages. A receipt of a message by an object triggers the execution of an operation, depending on the synchronization constraints of that object. Languages such as POOL-T, ABCL/1 and Hybrid take this approach.

A third possibility is to incorporate both active and passive objects. Active objects provide concurrency and synchronization, whereas passive objects are basically data abstractions that do not synchronize concurrent requests. Typical examples for this approach are Act++, Argus [Liskov 87], Eiffel// and PAL [Bjornerstedt 88].

2.2.2. Message-Passing Semantics

Since objects can only interact by sending messages, message passing is the basic means for creating concurrency and synchronization among objects. *Asynchronous* and *synchronous* message passing are the two fundamental message-sending constructs.

In the case of asynchronous communication, the sender object will not be blocked, even if the receiver is not ready to communicate. The sender is free to perform other computations after the message has been sent. An asynchronous message-passing model is used in ABCL ([Yonezawa 86], [Yonezawa 90]), Act-1 [Lieberman 87] and SR [Andrews 81].

In case of synchronous message passing, the sender object trying to send a message to another object is blocked until that object is ready to communicate. This synchronization model is used in Communicating Sequential Processes (CSP) [Hoare 78]. In the absence of any message buffering it is obligatory to have this mode of communication.

Many languages extend and mix these two communication policies to create different concurrency and synchronization semantics. We will study the following four constructs: *proxies*, *coordinated termination*, *remote procedure calls*, and *early returns*.

In case of proxies, the sender object sends its request asynchronously, and continues with its processing. The result of invocation of the operation is collected by a so-called proxy object. Whenever the sender object requires the result, it can then retrieve it from the proxy object. In case the result is not yet collected by the proxy object, the request to the proxy object is blocked. The *future* mechanism of ABCL, and ConcurrentSmalltalk ([Yokote 87], [Okamura 91]) are examples of this approach. The advantage of this model is that it increases the amount of concurrency in the system, without increasing the burden for the programmer.

Coordinated termination allows the sender object to send (asynchronous or synchronous) messages concurrently, but only continue with its processing when all these messages have been executed. The languages Orient/84 [Tokoro 86] adopt this model.

In the (blocking-) remote procedure call model, the sender object is blocked until the receiver completes the requested operation and returns the result to the sender object. This model further restricts synchronous communication in that the sender and receiver objects cannot be concurrently active. Such a model is useful in systems based on the request-reply paradigm of

computing. This model has been used in Ada [Ada 80], Argus [Liskov 87], DP [Brinch-Hansen 78], POOL-T [America 87], and SR.

In *early returns*, the sender object sends a message synchronously and is blocked from carrying out further operations. The receiver object can execute a *reply* statement somewhere during its execution. In this case, a result is returned to the sender object and both objects continue with their executions concurrently. The difference between early return and remote procedure call is that in early return, the receiver object can remain active after executing the return statement, whereas in remote procedure call, the receiver object terminates further execution. Ada provides an early return mechanism.

2.2.3. Creating Concurrency

Clearly, all message-send constructs except remote procedure calls can create multiple execution threads. Functionally, however, these constructs are not mutually exclusive. For example, early returns can emulate the simple asynchronous message send, if the receiver object executes the *return* statement immediately.

Objects can also create multiple threads by creating a new object. If the newly created object has its own *initialization* process, then both the sender and the instance object can be concurrently active. An initialization operation does not require an external message to be triggered. This way of creating concurrency is adopted for example by POOL-T [America 87].

2.2.4. Accepting Messages

To execute a request, first the receiver object must *accept* the received message. Accepting a message means that the corresponding operation for this message is to be initiated. This approach is also referred to as *interface control*. Languages adopt different mechanisms for accepting messages. The major approaches are *unconditional acceptance, explicit accept statements*, and *condition-based acceptance*.

In unconditional acceptance, the receiver object does not synchronize the received messages; it immediately executes them. This mechanism is adopted by Ada *packets* [Ada 80], Emerald [Black 86], Smalltalk [Goldberg 83], and Trellis/Owl [Schaffert 86].

Some languages adopt an explicit accept statement to execute a message. Until the *accept* statement is explicitly executed by the receiver object, the received messages are queued. Languages Ada, CSP [Hoare 78], Eiffel// [Caromel 90], POOL-T [America 87] and SR [Andrews 81] take this approach.

Condition-based acceptance is specified by observing the current state information of the receiver object. The state specification, specified either by referring to the explicit object variables, or an abstraction thereof, determines the condition that must be satisfied before a received message is accepted for execution. Typical examples of this approach are the languages ABCL/1 ([Yonezawa 86], [Yonezawa 90]), Act++ ([Kafura 89], [Kafura 90]), CEiffel [Loehr 92], Dragoon [Atkinson 91], Guide [Decouchant 89], Hybrid [Nierstrasz 87], PROCOL [Bos 89] and Rosette [Tomlinson 89].

There are different approaches to implementing condition-based acceptance.

For example, objects in Eden are constructed using Concurrent Euclid [Holt 83] which provides the monitor construct for achieving synchronization. In Path Pascal [Campbell 80] and PROCOL [Bos 89] (extended) path expressions are the mechanisms used for synchronizing access to an object.

Most languages introduce some sort of a *logical guard* operation to determine whether the requests are to be accepted, delayed or rejected. For example, DP and SR use Boolean expressions for synchronization. In Guide, each method is controlled by a condition that refers to the object's local variables and *synchronization counters* ([Gerber 77] and [Robert 77]). In Act++, however, the conditions are expressed as an abstract state, with which a set of acceptable methods is associated.

For the actor-based model [Agha 88], a mechanism called *serializer* [Atkinson 77] was introduced for synchronization. All messages to an actor (which represents the shared resource object) are relayed through its serializer, which enforces synchronization constraints.

As a result of these synchronization mechanisms, an object may carry out either a single operation at a time, a number of concurrent operations or a number of interleaved operations over a period of time. Executing only a single operation at a time (i.e. mutual exclusion) may be enforced by explicit or implicit guards.

Languages that allow several operations to be executed simultaneously are supported by guards that allow more than one request to be accepted at a time. Examples are Guide, Serializers, Ada *packets* and POOL-T *units*. Ada and POOL-T differ from other languages in that *packets* and *units* are not treated as objects. The third approach allows acceptance of more than one request, however only one request may be executed at a given moment. Interleaving executions is also possible. The language Hybrid adopts this approach.

3. Problems with Concurrent Object-Oriented Models

3.1 What is an Inheritance Anomaly

In section 2.2.4, we briefly studied the ways an object can accept the received messages. The synchronization constraints of an object are defined as a part of its specification and implementation. Several researchers have indicated (e.g. in [America 87], [Nierstrasz 91] and [Kafura 89]) that introducing a new method and/or overriding an inherited method in a subclass may require additional redefinitions. Ideally, this should not be necessary. This problem is referred to as *inheritance anomaly* [Matsuoka 90,93].

These anomalies originate mainly from two aspects of current object-oriented languages. Firstly, the synchronization and inheritance mechanisms of current object-oriented languages are often not very well integrated. Secondly, the power for expressing synchronization conditions to capture a certain situation is insufficient in most languages. These two aspects are studied in sections 3.2 and 3.3, respectively.

3.2. Conflicts between Inheritance and Synchronization

The conflicts between inheritance and synchronization can be divided into three categories, which are presented in the three following subsections.

3.2.1. Mixing Synchronization Code with Application Code

Mixing synchronization code with application code makes the employment of inheritance mechanisms into concurrent object-oriented languages very difficult. This mixing makes it impossible to change synchronization without affecting the application code, and vice versa. We will give two examples of this anomaly.

Some concurrent languages allow each object to have a part called the *body*; code that is executed independently from the execution of incoming messages (e.g. POOL-T [America 87], Eiffel// [Caromel 90]). The body may implement some general house keeping operations as well as the synchronization constraints of its object. One of the difficulties with this approach appears when additional synchronization semantics are required in subclasses; in this case the whole body, including the parts that are not related to synchronization, must be totally re-programmed.

Another example of this anomaly is explicit message reception. Some languages (e.g. Ada, ABCL/1) provide the programmer with facilities to put explicit message reception statements anywhere in the application code. The drawback of explicit message reception is that when the programmer extends an existing synchronization constraint through subclassing, the application code must be redefined. Implementation of the synchronization code within separate operations does not provide a complete solution to the problem because the application code still needs to make explicit calls to those synchronization operations.

3.2.2. Key Specifications

We use the term *key* specification[3] when within the implementation of a class, identifiers are used to directly denote some entity, for instance a method. In this discussion, we are only concerned with the usage of keys for entities involved with synchronization within the implementation of an object's operations. When keys are used in a class as part of the synchronization specification, this will result in anomalies when trying to extend such a class.

Key specifications may cause inheritance anomalies when a new entity is introduced or an entity is redefined in a subclass. If a new entity is introduced in a subclass, then this must be accounted for in several places throughout the class. When an entity is redefined in a subclass, then all the references to this entity are to be redefined accordingly. We will explain this by some concrete examples.

Specification of method keys is used by some actor-based programming languages which allow the programmer to define the set of messages that are acceptable after the current execution is terminated [Kafura 89]. This set of messages is called the *next accept set*.

---

3 The term *direct key specifications* was introduced in [Matsuoka 90], where it refers to the use of message keys in synchronization specifications.

Typical application code contains a series of *if-then-else* statements at the end of every method which specify the next accept set for each state of the object.

Consider, for example, a stack of limited size which offers two operations, *push* and *pop*, for data storage and retrieval, respectively. In case the stack is empty, the *pop* requests should be delayed until additional data is stored. Conceptually, the state of the stack can be divided into three states: *empty*, *partial*, and *full*, depending on the number of elements stored in the stack. When the stack is *partial* it offers both storing and retrieval services. However when it is *empty* it offers only the storing service. Similarly when the stack is *full*, only the *pop* operation is allowed.

In the next accept set approach, the following if-then-else statement could be defined:

```
if (numberOfElements = 0) then  accept {push}
elseif (numberOfElements < size) then  accept {push, pop}
elseif (numberOfElements = size) then  accept {pop};
```

When a new service is added in a subclass, the programmer has to modify the if-then-else statement to include the newly added operation in the proper accept sets. Such a modification is clearly not desired since it requires the redefinition of (possibly all) methods.

The same anomaly occurs in languages that adopt explicit *accept* [America 87] statements, due to the direct reference to method keys.

To avoid the problem of method key specification, some researchers introduced languages that incorporate *states* explicitly ([Kafura 89], [Tomlinson 89]). A state is referred to by its key; this concept is also referred to as *behavioral abstraction* [Kafura 89]. An important property of behavioral abstraction is its independence from a particular implementation, which improves the reusability of synchronization code. Using this concept, the synchronization for the stack example could be specified by the following code:

```
if (numberOfElements=0) then become empty
elseif (numberOfElements<size) then become partial
elseif (numberOfElements=size) then become full;
```

Instead of the method keys that were specified in the previous example now the keys of states are specified in the code. A *behavior* statement is provided to give a name to a group of message keys which correspond to conceptual state subspaces:

```
behavior: empty={push}, partial={push, pop}, full={pop}
```

When a new operation is added in a subclass the only required change is the insertion of this new operation into a proper state subspace. This can be accomplished by redefinition of the behavior declaration only. Therefore the problem as explained for simple accept sets does not occur in this approach. However, problems will arise when a class is extended in such a way that a state is divided into substates. All the locations were the key of this state is specified have to be updated, in order to account for the substates.

As an example of a state key anomaly, consider the addition of a new operation, *pop2*, to the stack. The operation *pop2* retrieves two elements from the stack, and it divides the state *partial* into two subspaces, respectively *oneElement* and *moreThanOneElement*. As a consequence, the stack implementation must be modified:

```
if (numberOfElements=0) then become empty
elseif (numberOfElements=1) then become oneElement
elseif (1<numberOfElements<size) then become moreThanOneElement
elseif (numberOfElements=size) then become full;

behavior: empty={push}, oneElement={push, pop},
          moreThanOneElement={push, pop, pop2}, full={pop, pop2};
```

Other examples of languages suffering from variations of the key specification anomaly are Hybrid [Nierstrasz 87] and Sina/st [Tripathi 88a].

3.2.3. Non (De-)Composable Synchronization Specifications

Synchronization specifications of some languages are monolithic units that cannot be further decomposed. As a consequence, it may be difficult to add new synchronization constraints or to redefine part of the synchronization specification in a subclass. The requirement of 'compositionality' was also identified in [Papathomas 91].

Consider for example path expressions [Campbell 74]. Extended path expressions [Andrews 83] can conveniently express most synchronization constraints. A path expression defines possible message acceptance sequences. A basic requirement for applying inheritance is the (de-) composability of synchronization constructs. Subclasses must be able to add or override new synchronization constraints. Path expressions thus do not fit into object-oriented concurrent programming due to poor (de-)composability.

Languages suffering from lack of decomposability are for example PROCOL [Bos 89], which has a mechanism that is a variation of path expressions, and languages that use 'bodies' for synchronization, such as POOL [America 87] and Eiffel// [Caromel 90].

Composition of synchronization constraints from multiple superclasses can be equally difficult. For example, consider class *LockingStack* which inherits from classes *Stack* and *Locking*. Class *LockingStack* is a stack which can be locked and unlocked. If the stack is locked no methods are accepted for execution, except the method *unlock* that removes this restriction.

The synchronization constraints that are defined by the two classes are sometimes contradictory. For instance, suppose an instance of *LockingStack* is in the locked state, and it contains no elements. Class *Locking* dictates that all methods except *unlock* are to be buffered. However, according to class *Stack*, for instance the method *put* should be acceptable.

In general, combining such contradicting synchronization constraints is not trivial.

3.3. Lack of Expressive Power for Synchronization Conditions

Due to the lack of expressive power for synchronization conditions, the programmer may be forced to write application code (i.e. one or more methods) which is actually implementing the synchronization conditions. This results in anomalies similar to those described in section 3.2.1.

In order to illustrate this, consider the following example. A stack is extended with an additional retrieval operation called *popAgain*, which is only to be executed right after an

invocation of a pop operation. In general, there are no means to express this condition, because it depends on *history information*. that is not available within the specification of synchronization conditions.

One way to resolve this problem is to extend the implementation to record information such as the past message acceptance order. This can be represented by additional instance variables. However, this requires the methods *push* and *pop* to be redefined in order to incorporate this functionality[4].

In general, synchronization conditions should be expressible in terms of any aspect of the state (in an abstract sense) of an object. This includes obvious aspects such as the instance variables of an object, or the arrival order of messages. An example of the latter are readers/writers with equal priority for the readers and the writers; an incoming reader message is not allowed to be executed when a writer message has been received previously (for example in DRAGOON [Atkinson 91] it is not possible to express this). Another important aspect for synchronization, which is not supported by many languages, is the values of the arguments of a received message ([Nierstrasz 91], [Liskov 87]).

3.4. Requirements for Concurrent Object-Oriented Languages

From the discussion in the previous sections, we infer several requirements[5] which object-oriented concurrent languages should fulfil, in order to be considered suitable for constructing extensible and reusable concurrent applications.

Modular concurrency:

There are three basic requirements for modular concurrency. Firstly, modularity must be supported at every level. A pure object-oriented model only consists of modular entities called objects; data structures that represent the objects in the *real-world*. Secondly, each object must be considered to be a potentially active entity. Many activities in the real-world are concurrent. To model the concurrent behavior of the real-world, programming languages must provide a means for decomposing a large computing task into several concurrently active objects. Thirdly, each object must determine its own synchronization semantics.

Expression-power:

Synchronization constructs of a language must be general and powerful. A language should provide a set of basic but powerful concurrency and synchronization constructs, on a per object basis that can be conveniently applied to build *tailored* concurrency and synchronization constructs. In addition, concurrent languages must be able to express synchronization conditions in terms of for example history information, or message content (as described in section 3.3).

---

4 Note that this cannot be completely solved by redefining a method which calls 'super', since this is not fully open-ended (often all the methods of a class must do some bookkeeping, even the newly added methods in subclasses).

5 Our list of requirements is not identical, but complies with the requirements on synchronization mechanisms as stated in [Bloom 79], [Grass 86] and [Nierstrasz 91].

Internal concurrency:

Real-world objects are characterized by internal object concurrency. Therefore, languages that allow for concurrency within an object would be more expressive.

Data-consistency:

A concurrent system designer has to struggle between two conflicting goals: to maintain the information stored in the system in a correct and valid state, while providing maximum accessibility to information, including the ability to refer to, delete or update the data. Concurrent executions can cause inconsistencies if one activity refers to data while another one is modifying it. Therefore, concurrent programming languages must provide synchronization mechanisms to enforce data consistency.

Free from conflicts between inheritance and synchronization:

Ideally, languages must be free of the inheritance anomalies that are presented in section 3.2.

Efficiency:

The mechanisms introduced by a language must be suitable for efficient implementations, in that the synchronisation mechanism should not incur too much overhead.

4. Our Approach: The Composition-filters Model

There are obvious benefits to using the traditional object-oriented model, providing encapsulation, data abstraction, classes and inheritance [Wegner 87]. However, this object model is not powerful enough to fulfil the concurrent processing and synchronization requirements as explained in section 3.4. We developed the concept of *composition filters* as an extension to the traditional model to encompass these requirements. We will present the composition-filters object model in this section. This computation model is adopted by the Sina language[6].

4.1 The Basic Object Model

We will first briefly introduce the components of the composition-filters object model, and then present them in greater detail later.

As illustrated by figure 1, a composition-filters object is subdivided in two parts: an *interface* and an *implementation* part. The interface part deals with incoming and outgoing messages, which are handled by input- respectively output-filters. The latter is not dealt with in this paper, input filters will be described extensively in this section. The implementation part contains operation definitions, local variable declarations, definitions of conditions, and an

---

[6] The early version of the Sina language was published in [Aksit 88], [Tripathi 88a] and [Aksit 91]. This version introduced only a simple filter mechanism. Although the expression power of the concurrent processing and synchronization mechanisms fulfilled our requirements, it suffered from inheritance anomalies. This paper extends the early version to address these problems.

optional initialization operation. The implementation part is fully encapsulated within the object. In the Sina language, operations and local variables are called methods and instance variables, respectively.

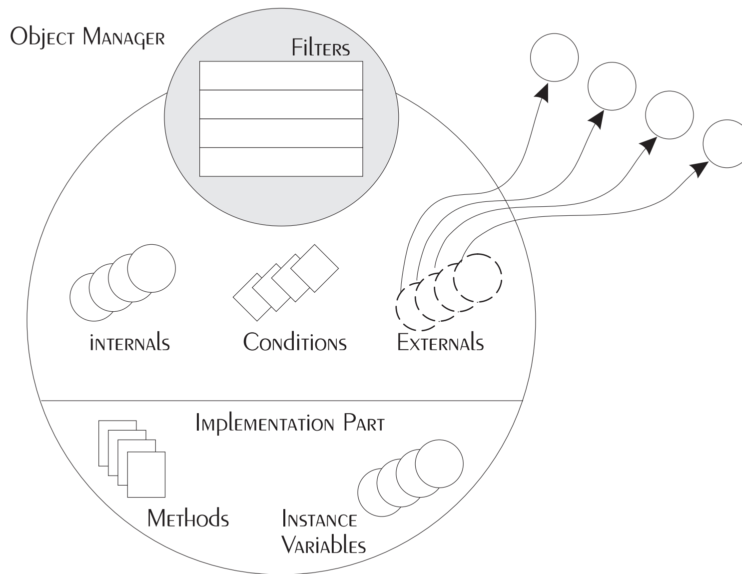


Figure 1. The components of the composition-filters object model.

4.1.1. The Interface Part

As an example of a simple class, consider the interface part of class *OrderedCollection*. We present our examples following the Sina language notation.

```
class OrderedCollection interface
    comment This class implements an ordered collection of Elements;
    methods
        add(Element) returns Nil;          // adds an element to the collection
        at(Integer) returns Element;       // gets an element at given index
        remove(Integer) returns Nil;       // removes element at given index
        size returns Integer;              // returns the number of elements
    conditions
        Initialized;                       // this condition is valid when the object has been initialized
    inputfilters
        < disp : Dispatch = { Initialized => * }; >
end;
```

Figure 2. Definition of the interface part of class *OrderedCollection*.

The *methods* that are to be visible at the interface of the object are declared in the interface part by *method headings* following the keyword "methods". Class *OrderedCollection*, for instance, declares the methods *add*, *at*, *remove* and *size* for manipulating an ordered collection of instances of class *Element*. The actual implementations of these methods are encapsulated within the implementation part. To invoke one of these methods, an appropriate message must be sent to an instance of class *OrderedCollection*.

A received message will only result in the execution of a method, after the message successfully passes all the *input filters* in the interface part of the object. An input filter specifies conditions for message acceptance or rejection and determines the appropriate

subsequent action. The output filters handle outgoing messages, and are mainly used to support the so-called Abstract Communication Types[7]. After the keyword "inputfilters", class *OrderedCollection* defines one input filter, called *disp,* of class *Dispatch* using the expression

```
disp: Dispatch = ....
```

An input filter of class *Dispatch* is used to initiate execution of a method if the message successfully passes it. Between the brackets "{" and "}", the filtering condition is specified as

```
{ Initialized => * }
```

On the left hand side of the characters "=>", a necessary condition is specified, which is denoted by the *condition* identifier *Initialized* in this case. Conditions are similar to logical propositions. The names of the conditions are declared in the interface part following the keyword "conditions", and their definition is provided in the implementation part. Conditions may reflect the values of instance variables, but may also reflect external variables. In this example, the condition *Initialized* depends on the value of the instance variable *initializedElements*, which is implemented as

```
initializedElements = 100;
```

Here, *size* and *initializedElements* are instance variables and the condition *Initialized* becomes *true* if all the elements of the ordered collection are initialized.

The received message is matched with the method names specified on the right hand side of the characters "=>". Here, the character "*" indicates a don't care condition; if the message matches with one of the method names provided by class OrderedCollection, then it will be accepted for execution. An alternative could be to list all the method names explicitly.

An optional *internal* clause may be used to declare nested objects, and an *external* clause may be used to declare exterior objects that are to be accessible to this object. Their use will be explained when inheritance mechanisms are introduced in section 4.2.

4.1.2. The Implementation Part

The components of the implementation part are exemplified by class *OrderedCollection* as shown in the following template:

```
class OrderedCollection implementation
    comment This class implements an ordered collection. The elements are stored in an array;
    insvars
        collection: Array(100);
        noOfElements: Integer;
        initializedElements: Integer;
    conditions          // the conditions that were declared in the interface part are implemented here
        Initialized
            begin  return initializedElements=100;  end;
    initial
        begin ...  end;   // here the initial method is defined, which is executed just after object creation..
```

7 Abstract Communication Types are used to abstract and reuse communication patterns between objects. More information can be found in [Aksit 89], [Aksit 92].

```
    methods
        add(anElement: Element) begin .... end;
        at(index: Integer) begin .... end;
        remove(index: Integer) begin .... end;
        size begin .... end;
end;
```

Figure 3. Template of the implementation part of class *OrderedCollection*.

Instance variables are declared in the *insvars* clause. Instance variables are fully encapsulated and can be objects of arbitrary complexity. Class *OrderedCollection* declares three instance variables named *collection, noOfElements* and *initializedElements*. The instance variable *collection* is used to store elements of the collection, and the other two variables are used for bookkeeping purposes. Only the methods defined within the object's class may access the instance variables directly; external clients of an object or even its subclasses cannot do this.

Subsequently, the implementations of the conditions are defined by message expressions. The structure of a condition implementation is similar to the structure of methods. However, a condition implementation always results in a Boolean value, and is free of *side-effects*.

The initialization method of an object is defined in the *initial* clause. This method is executed immediately after object creation.

The last component of the implementation part is the definition of the methods. A method consists of a series of message expressions and their sequence may be controlled by a set of standard *control statements*.

4.1.3. Inter-Object Communication

In the Sina object model, invocations are based on messages using the request-reply model of communication. An object can send a message to another object by using the receiver object's name. Access to an object is restricted to the scope of its name. An example of a message expression is the invocation:

```
aCollection.add(anElement);
```

This results in sending a request message *add(anElement)* to the object *aCollection* which is an instance of *OrderedCollection* described in figure 2. *aCollection* here is the receiver object, *add* is the method to be invoked, and *anElement* is the message argument. The receiver object is also termed as the *target* of the message, and the method name as the *message selector*.

After sending a request message, the sender will be blocked until it has received a reply. An invoked method can return an object to the sender of the message by using the expression

return *anObject* ;

When no result is explicitly returned, the method will return an object, which is an instance of class *Nil*. This is an early return statement as described in section 2.2.2; after executing the *return* statement the method may continue processing if there are any statements following the *return* statement.

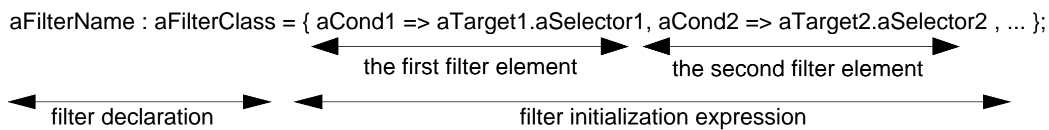
The Sina compiler incorporates a pre-processor that allows programmers to use a more familiar short-hand notation for messages such as assignment, arithmetic and logical operations. For example, the addition of two numbers *a* and *b* may be denoted by "*a.plus(b)*",

but also by "*a + b*". In the latter case, the pre-processor converts the expression to the standard message sending form *a.plus(b)*.

4.1.4. Message Acceptance

A filter is a *first-class* object that determines which action must be taken when a message is *accepted* or *rejected*, respectively. Each filter is declared as an instance of a filter class. A programmer may define an arbitrary number of filters for an object. Each filter can be an instance of an arbitrary filter class. The complete set of input filters of an object determines the conditions for message acceptance, and determines which method will be executed upon acceptance.

A filter must be initialized by a *filter expression*, which is shown at the right hand side of the "=" character in the following template:



Here, the filter *aFilterName* is declared as an instance of class *aFilterClass*. A filter is initialized by a sequence of filter elements separated by commas. A filter element defines a specific condition for accepting a particular set of messages. As shown above, each filter element consists of a *condition identifier*, and a *target-selector pair.*

When a message is to be evaluated by a filter, it will be checked against the elements of the filter, in left-to-right order. The evaluation of a single filter element consists of the following two conditions:

(1) *Condition evaluation*: First the condition of the filter element is evaluated. If the condition evaluates to *true*, the next step (matching) will be carried out. If the condition evaluates to *false*, then the filter element is skipped.

(2) *Matching*: First, the selector of the received message is matched with the selector specified by the filter element. If this match is successful, then the target specified by the filter element is bound to the message. If the match operation fails, then the filter element is skipped.

When these conditions are satisfied, the message is said to be *accepted* for this filter. In case the filter is of class *Dispatch*, this will result in delegation of the request message to this target object. If none of the filter elements satisfies the two conditions, then the message is rejected by this filter. The class of the filter determines the specific rejection behavior. Typically, rejection will result in an error.

Another type of filter is class *Error*. This filter class is used for the selection and rejection of messages only. When a message is rejected by the filter it raises an error condition. This causes the abortion of the received message. When the message is accepted it will proceed to the next filter.

Every arrived message is subject to evaluation by the complete set of input filters, until it is either dispatched to the appropriate method or discarded due to an error or exception condition. An accepted message will be dispatched eventually, resulting in execution of the

corresponding method. Since a message must pass all filters in declaration order, an AND condition between the filters is realized.

The basic mechanism as presented here has some extra features, most of them for convenience:

- If no condition is explicitly specified in a filter element, then the condition *True* is assumed. For readability purposes, the programmer may also write the condition *True* explicitly.
- Instead of using a name to indicate a target or a selector, the character "*" may be used, which denotes a don't care condition.
- If the target part is omitted, the pseudo-variable *self* is assumed.
- To shorten filter expressions, one can combine several filter elements together.

  For example, instead of a single condition, a set of conditions can be provided, as follows:
  ```
  {condition1,condition2}=> aTarget1.aSelector1
  ```
  is equivalent to
  ```
  condition1=>aTarget1.aSelector1, condition2=>aTarget1.aSelector1
  ```
- Another example is to use a short-hand notation when a single condition corresponds to several target-selector pairs:
  ```
  { condition1=>{aTarget1.aSelector1, aTarget2.aSelector2} }
  ```
  which is equivalent to
  ```
  { condition1=> aTarget1.aSelector1, condition1=>aTarget2.aSelector2 }
  ```

4.2. Data Modeling Techniques Through Input filters

This section demonstrates how input filters can be applied to basic object-oriented data modeling techniques, such as inheritance and delegation. In section 4.3 we will explain how filters can be used to define concurrency and synchronization mechanisms.

In the composition-filters model, inheritance is not directly expressed by a language construct, but is simulated by input filters. In order to inherit from a class, an *internal object* must be declared as an instance of that class. By delegating messages to the methods provided by this instance object, inheritance is simulated. This is exemplified by class *Stack*:

```
class Stack interface
    internals
        coll : OrderedCollection;          // instance of the 'superclass'
    methods
        isEmpty returns Boolean;
                                           // returns true when the stack contains no elements
        pop returns Element;               // gets and removes the top element
        push(Element) returns Nil;         // adds a new element at the top of  the stack
    inputfilters
        < disp:Dispatch= { True=>coll.*, True=>inner.* }; >
end;
```

Figure 4. Definition of the interface part of class *Stack*.

Class *Stack* declares an internal object *coll* of class *OrderedCollection* and introduces three methods *isEmpty*, *pop* and *push*. The method *isEmpty* returns *true* if the *stack* is empty, otherwise returns *false*. The method *pop* removes the *top element* from the stack and returns

an instance of class *Element*. The class also provides a method *push*, which takes a new element as an argument and adds this element on top of the stack.

The filter *disp* of class *Dispatch* contains two filter elements. The condition *True* preceding each filter element means that the target-selector pair(s) on the right-hand side will always be checked. These two filter elements have the following meaning:

First filter element:

> The first element of the filter, "coll.*", means that all the incoming messages are delegated to the internal object *coll*, provided that these messages are supported by class *OrderedCollection*.
>
> Since the methods of *OrderedCollection* are also available to *Stack* through an instance of *OrderedCollection*, class *Stack* inherits the operations of class *OrderedCollection*. The clients of the instance of *Stack*, however, do not observe that these methods are provided by an instance of class *OrderedCollection*. For example, a client can send the message *size* to object *aStack*, which is an instance of class *Stack*, where *size* is inherited from *OrderedCollection*. This inheritance mechanism is also referred to as delegation-based inheritance.
>
> When an instance of class *Stack* is created, then its internal object *coll* is first created. The internal object *coll* stores the elements of the ordered collection. Obviously, each instance of class *Stack* will have its distinct collection of elements. This is equivalent to inheriting instance variables of a super class. An important feature here is that instance variables of the super class are only accessible through operations provided by the super class [Snyder 86].

The second filter element:

> If the first filter element does not match with the message, then the second filter element is evaluated. Instead of delegating to an internal object such as *coll*, this filter element delegates the message to the pseudo-variable *inner*. By declaring *inner* as a target object, class *Stack* makes the methods defined and implemented by itself available to its clients.

Notice that since the filter elements are evaluated from left to right, therefore the first element prevails over the second one. Note that the order of the filter elements can be manipulated to bind messages to the desired targets. This is for instance applicable for solving name conflicts.

Instead of using an internal object as a target, the programmer can also delegate the incoming messages to an *external object* by declaring the target name in the *externals* clause. This is very similar to delegation to internal objects, however the external object is not encapsulated within the delegating object. This means external objects can be shared by other objects. In addition, contrary to the *internals* clause, this declaration does not result in automatic object creation.

Apart from the pseudo-variable *inner*, two other pseudo-variables, *self* and *server*, are also available as a means of self-reference. The distinction between these three pseudo-variables is exemplified by the following figure:

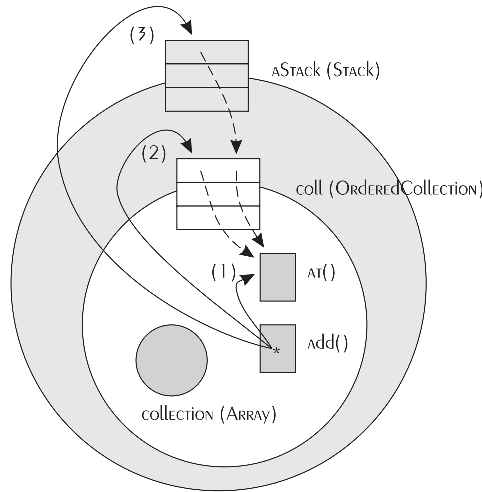


Figure 5. Representation of an instance of class *Stack* and the projected message flows for *inner.at()* (1), *self.at()* (2) and *server.at()* (3).

Figure 5 depicts an instance of class *Stack*, called *aStack*, encapsulating the internal object *coll*, which is an instance of class *OrderedCollection*. When a message "aStack.add(anElement')" is invoked by another object (the client) it will be delegated to *coll* by the filters of *aStack*, after which it passes through the filters of *coll*, which results in the execution of method *add()* as provided by object *coll*. Suppose that during the execution of method *add()*, it is required to invoke the message *at(1)* upon the object itself. There are three options for achieving this:

1. By using "inner.at(1)", the method *at()* as defined by *coll* is invoked directly; the message does not have to pass any filters. In figure 5, this is shown by message flow (1). Thus, *inner* could also be used for calling methods that are not visible at the interface of an object.
2. When the expression "self.at(1)" is executed, the message "at(1)" will be sent to the interface of the object executing the expression, which is *coll* in this example. The message is then processed by the input filters of object *coll*. In this case, the filter evaluation will result in a dispatch to the local method *at()*, as shown by message flow (2) in figure 5. Note that method *at()* must be visible at the interface of object *coll*. Therefore, in order to invoke an inherited method, *self* must be used as the target of the message.
3. Invoking message "server.at(1)" will cause the message *at()* to be sent to the receiver of the message *add()*, which is the object *aStack* in this example. The input filters of *aStack* then determine how to dispatch the message; in this example it will be delegated to object *coll* (message flow (3) in figure 5). The term *server* refers to the object *aStack* that is serving the request of the client object. Using *server* realizes dynamic binding in that the method *at()* may be overridden by a new method *at()* defined by *aStack*, whereas messages to *inner* and *self* are statically bound.

A fourth pseudo-variable, *sender*, is available, which provides a reference to the sender of the message that is currently being processed.

Typically, in most object-oriented languages every class inherits -either directly or indirectly- some default behavior from a root class called *Object*[8]. The Sina system contains a primitive class called *Object* which provides the default operations for all the classes. Typical examples of default operations used in this paper are *assign*, *equal*, and *copy*.

The Sina compiler provides an option to add an instance of class *Object* called *default* to the internals, and automatically insert "True=>default.*" as the first filter element of every filter. This option eliminates the necessity for programmers to define the default operations explicitly for every new class. Since *default* is the first element of a filter, it prevails over other filter elements. Programmers can explicitly override this option and create an internal object *default* which is an instance of a customized version of class *Object*. This object, for example, could eliminate the *assign* operation of *Object* so that a constant behavior within the class can be assured.

4.3. Synchronization Through Filters

In this section we will first explain how concurrency is created and managed in the composition-filters model (section 4.3.1), then it is explained how synchronization constraints are expressed (section 4.3.2). In section 4.3.3, the object manager is explained. Finally, recursive messages (section 4.3.4) and default filters (section 4.3.5) are discussed.

4.3.1. Concurrency-Control Mechanisms

There are two ways to achieve concurrent threads. Whenever an object is created, its (optional) initial process is activated, which may remain active for an arbitrary length of time. Another means of achieving concurrency is by using early return statements, as presented in section 4.1.3.

The concurrency model of the Sina language supports internal object concurrency because the language does not restrict the number of active threads within an object. Consistency of an object can be guaranteed by defining the appropriate filters. By default, the pre-processor includes for every object a filter which provides mutual exclusion, allowing only one active thread within the object. Mutual exclusion ensures that no inconsistencies will occur in the values of the instance variables. By specifying a compiler option, the default filter for mutual exclusion is not included, and less restrictive concurrency constraints can be specified.

Every object has an *object manager*, which is a system-defined object that schedules the processing of request messages to the object it manages. An object manager receives messages, places them in the message queue, initiates filter evaluations, and removes the message from the queue when it is activated. In addition, an object manager keeps information about the active threads.

When a message arrives at an object, its object manager places the message at the end of the message queue. Putting the message in the queue is an indivisible operation; only one

[8] 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*.

message can be queued at a time. A message remains queued until it is either rejected or accepted.

Two important properties are ensured by the object manager:

(1) Evaluation of the set of filters is done atomically[9], thereby ensuring that all the filter constraints are satisfied for the received message before the message is dispatched.

(2) When two messages are subject to the same constraints, assuming that both messages are eventually dispatched, the message that is closer to the head of the queue will be dispatched first. This does not mean that messages are always executed in first-come-first-served order. A message that arrived earlier but does not yet fulfil the conditions imposed by the filters will allow the execution of another message, if that message does fulfil its conditions.

4.3.2 Expressing Synchronization Constraints

Synchronization constraints are specified by filters of class *Wait*. When a message is accepted by a filter of class *Wait*, the message passes to the next filter. If, however, the evaluation is not successful, the message remains in the queue until it fulfils the condition of one of the filter elements. Requests to methods of an object can be synchronized by associating messages with specific conditions that implement specific synchronization conditions.

When a condition associated with a *Wait* filter element evaluates to *true*, a message that was previously blocked on this condition may now be accepted and can pass to the next filter. As an example, consider class *SyncStack*:

```
class SyncStack interface
    comment inherits from class stack, and adds a synchronization
            constraint, i.e. a pop message to an empty stack will be
            blocked until there is an element in the stack;
    internals
        inStack:Stack;
    conditions
        NonEmpty;      // true when there is at least one element in the stack;
    inputfilters
        < sync:Wait={NonEmpty=>pop, True=>*\pop };   // specifies synchronization constraints
          disp:Dispatch={ inStack.* }; >   // provides all inherited methods from Stack
end;
```

Figure 6. Definition of the interface part of class *SyncStack*.

Class *SyncStack* extends *Stack* by adding synchronization constraints. If the stack is empty, the condition *NonEmpty* will be false, therefore a request to the method *pop* will be blocked. This is expressed by the first filter element of filter *sync*. In the second filter element, the expression "*\pop" is used to indicate that all messages are acceptable excluding message *pop*. Thus, messages *push*, *at*, *remove* and *size* will always pass this filter, as they are associated with the condition *True*. Note that class *SyncStack* inherits from *Stack*, making all the methods of class *Stack* available at the interface of *SyncStack*.

---

9 In this paper we consider the evaluation of a filter-set to be done atomically. This is actually specified explicitly by the brackets "<" and ">" surrounding the filter declarations. Although non-atomic evaluation allows for more efficient synchronization specifications, for the sake of consistency we use atomic filter-sets for all examples in this paper. In some cases, atomic evaluation is necessary to guarantee the indivisibility of the filter-set evaluation.

Implementation of the condition *NonEmpty* is defined as follows:

```
NonEmpty
    begin return inStack.size>0; end;
```

The condition value is determined by the size of the internal object *inStack*. The method *size* is provided by class *Stack*.

In order to demonstrate synchronization of class *SyncStack* , we show the message queue for an instance of class *SyncStack* that subsequently receives the messages *add*, *pop*, *pop'*, *pop"*, and *add'* in figure 7.

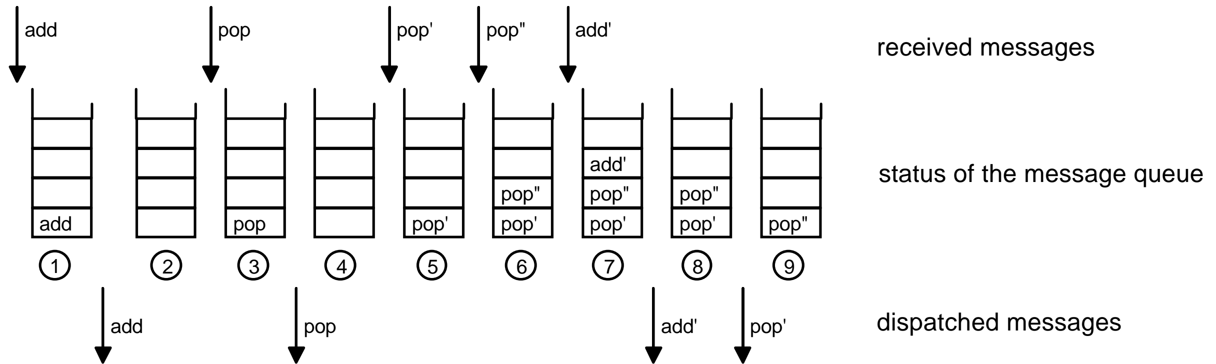


Figure 7. Message queue for an instance of *SyncStack*.

As the first message *add* is received (no. 1), it is immediately dispatched and therefore removed from the queue (no. 2). Now the stack contains one element, and condition *NonEmpty* evaluates to *true*. So when message *pop* arrives (no. 3), it is accepted by the *sync* filter, and immediately dispatched (no. 4). Now the condition *NonEmpty* is no longer *true*; the message *pop'* remains in the queue (no. 5). When the message *pop"* subsequently arrives, it is placed in the queue after *pop'* (no. 6). When *add'* arrives, it is placed at the tail of the queue (no. 7), but since it is the first acceptable message in the queue, it will be dispatched prior to the *pop* messages (no. 8). After *add'* has completed its execution (mutual exclusion is enforced through a default filter, as mentioned in section 4.3.1), condition *NonEmpty* is valid once again. This enables both *pop'* and *pop"*, in which case the first applicable message (*pop'*) in the queue is dispatched. After the execution of *pop'*, the condition *NonEmpty* will be *false* again, and *pop"* remains in the queue (no. 9).

4.3.3. Accessing the Object Manager

Since every object manager is responsible for receiving, buffering and dispatching messages, it maintains information regarding the number of active and blocked message requests. This information can be obtained by sending messages to it. An object can send messages to its own object manager by specifying the identifier "^self" or "^server" as the target of a message invocation. Each object manager is encapsulated within its object, and cannot be

accessed by other objects[10]. Apart from monitoring the received messages, the object manager also provides support for returning results of message invocations.

returning results

An object manager is responsible for returning replies to objects that have sent a message; the statement "return anObject" that we have used previously, is actually a syntactic equivalent of "^self.reply(anObject);". The *reply()* is provided by the object manager and returns its argument to the sender, while the requested method may continue processing subsequent statements after the result has been returned.

monitoring

The object manager also provides methods for retrieving the values of synchronization counters [Gerber 77] [Robert 77]. The values maintained in the synchronization counters indicate the number of received messages, the number of dispatched messages, and the number of completed method executions. This is done both for the object as a whole, and for the individual methods.

The number of active threads[11] within the object, can be calculated by using the following expression:

```
^self.dispatched - ^self.completed;
```

For convenience, the object manager provides a method *active* which returns the number of active threads. Similarly, the method *blocked* returns the number of blocked processes, which could also be expressed as:

```
^self.received - ^self.dispatched;
```

Examples of synchronization counters are found in Guide [Decouchant 89] and 'Synchronizing Actions' [Neusius 91], and are especially useful for managing intra-object concurrency. As an example, we show how mutual exclusion can be defined by a filter:

```
conditions
    Free;
inputfilters
    mutEx : Wait = { Free=> * };
```

The condition *Free* indicates that currently there is no active thread within the object. It is implemented as:

```
Free
    begin return ^self.active=0 end;
```

The filter *mutEx* blocks all messages while the object is active in processing a request. After that request is completed, the first message in the queue will be evaluated by the filter. Note that before evaluating other messages in the queue, condition(s) will be updated, and thus *Free* will be invalid again.

---

10 An exception is through the usage of '^server', which refers to the object manager of the server object. We consider this not to be a problem, since the object which refers to ^server, is a delegated object, and thus "part of the extended identity" of the delegating ('server') object.

11 'Active' here means that a thread is executing some method; the message has been dispatched but has not completed its execution.

4.3.4. Recursive Messages

A recursive message is a message that is sent to one of the pseudo-variables *self*, *server* or *sender*. When an object enforces mutual exclusion and issues a recursive call within one of its methods, this would result in deadlock. The reason for this is that the recursive message would have to pass the filters of the object, which are blocked because there is already an active thread within the object. The preferred solution is that a recursive message would be immediately accepted by the filters.

Because mutual exclusion is not a part of the language, but defined by filters, all that is required to cope with recursive messages is a suitable filter specification. This filter must implement mutual exclusion for all messages except for recursive calls. The condition *Recursive*, implementing this condition, determines whether a message is recursive. The definition of the filter *mutEx* is shown in figure 8. This filter is provided by default for all objects.

```
conditions
    ... Free;    Recursive; ...
inputfilters
    ... mutEx : Wait = { Recursive=>*, Free=>* }; ...
```

Figure 8. The definition of a filter for mutual exclusion, resolving recursive messages.

4.3.5. Default Filters

If mutual exclusion is enforced within an object, no inconsistencies will occur due to methods concurrently accessing the same instance variables. Since mutual exclusion is desirable for most applications, filters defining mutual exclusion are provided through a compiler option. Figure 9 shows the internal object, the conditions and the filters which are inserted by the pre-processor for every object:

```
class .... interface
    internals
        default : Object;        // defines an instance of the class with the default behavior
        ...
    conditions                   // here the conditions that are reused from Object are declared:
        default.Initialized;     // is valid when the initial method is still active
        default.Recursive;       // is true when the message is recursive
        default.Free;            // is true when no method is currently executing within this object
        default.Protected;       // will be valid only when the sender is a delegated
                                 // object (subclass), i.e. server.contains(sender)
        ...
    inputfilters
        < default.initialization;    // block all methods when initial is active
          default.defMutEx;          // mutual exclusion
          default.defMethods;        // inherit the methods from class Object
        ... >
end;
```

Figure 9. Demonstration of the inclusion of the default object, conditions & filters.

In order to explain the default behavior of objects, and demonstrate that the filter construct can be used to define what could be considered as low-level behavior in a transparent way, the definition of the default filters and conditions dealing with synchronization is shown here and explained.

The filter *initialization* blocks the interface of an object while the *initial* method is still active. This prevents the execution of methods while the object is not completely initialized. Note that it is possible to change this filter, in order to allow the initial method to continue while the object accepts messages. The definition of the filter and the condition *Initialized* are as follows:

```
filter:      initialization:Wait= { Initialized=>* };
condition:   Initialized begin return ^self.activeFor(initial)=0; end;
```

The filter *mutEx* provides the default mutual exclusion mechanism, while allowing recursive messages, as explained in the previous subsection.

```
filter:      mutEx : Wait = { Recursive=>*, Free=>* };
conditions:  Recursive  begin  return message.isRecursive;  end;
                 // recursive messages are message that are sent to either self, server, or sender
             Free  begin  return ^self.active=0;  end;
```

The primitive objects in the system, such as integers, booleans and strings all provide a behavior that is equivalent to the behavior of the default filters. Thus, an application programmer is not necessarily concerned with synchronization issues, but can enforce mutual exclusion for all objects by inserting the default filters through a compiler option.

5. Examples

This section gives a number of examples in four categories. Some of these examples are defined as a pair of classes. The first class is the reference problem, and the second class is its subclass. The motivation for the second class is to illustrate the reusability and extensibility features of the composition-filter based approach. Some examples are given to demonstrate the expression power of the mechanisms in the area of object-oriented concurrent programming and synchronization.

5.1. Examples of Intra- & Inter-Object Synchronization

This section presents three examples, demonstrating both intra- and inter-object coordination. The first example consists of several extensions to the stack example we presented in the preceding text. The second example illustrates the use of arrays of objects to construct solutions based on parallel processing. The last example in this section describes a circulating token scheme, as used in distributed systems for implementing mutual exclusion between a number of objects.

5.1.1. Synchronization Extensions of Stack

We continue with the stack example as presented in figures 2-6, and introduce some further extensions.

The first example, as shown in figure 10, gives the definition of class *Pop2Stack*, which inherits from class *SyncStack* (that was defined in figure 6), and introduces a new method, *pop2*. The method *pop2* gets two items from the buffer at once, instead of a single item: this requires additional synchronization. Class *Pop2stack* synchronizes and implements the method *pop2* by calling the method *pop* twice and combining the results in a pair object. The synchronization filter *pop2Sync*takes care that no other *pop* message can be executed while a

pop2 is executing; this ensures that the two elements that are retrieved by the *pop2* are also subsequent elements from the stack.

```
class Pop2Stack interface
      internals
            superStack:SyncStack;
      methods
            pop2 returns Pair;
      conditions
            FilledWith2;
            NoPop2Pending;
      inputfilters
            < pop2Sync:Wait = { FilledWith2=>pop2, True=>*\{pop2} };
              disp:Dispatch = { superStack.*, inner.pop2 }; >
end;

class Pop2Stack implementation
      conditions
            FilledWith2  begin  return superStack.size>1; end;    // true when 2 or more elements in buffer
      methods
            pop2     // the method returns an instance of class Pair containing the 2 elements
                  objects p:Pair;   // declare a temporary object of class Pair
                  begin  p.putFirst(superStack.pop); p.putSecond(superStack.pop); return p;  end;
end;
```

Figure 10. The definition of class *Pop2Stack*.

The next example demonstrates the composition of objects into a new object through the 'locking' example. In order to show this, first the class *Locking* is defined, which is defined in figure 11. This class provides two methods, *lock* and *unlock*, which respectively 'lock' the object, causing no method except for the *unlock* to be accepted, and 'unlock' the object, causing all methods to be acceptable again. The status of the object is stored in a boolean instance variable, *free*.

```
class Locking interface
      methods
            lock returns Nil;
            unlock returns Nil;
      conditions
            Unlocked;
      inputfilters
            < locksync:Wait={ True=>unlock, Unlocked=>*  }; >
end;

class Locking implementation
      instvars
            free:Boolean;
      conditions
            Unlocked  begin return free; end;
      methods
            lock  begin free:=false; end;
            unlock  begin free:=true; end;
end;
```

Figure 11. The definition of class *Locking*.

The method *unlock* is always available, independent of the state of the object. The other methods of the object are only allowed when the object is the *Unlocked* state. Note that, due to the usage of the '*', this synchronization specification is open-ended, in the sense that the specification will still be valid when new methods are added (assuming these have to be blocked in the *lock* state as well).

Class *LockingStack* is a composition of class *Locking* and class *SyncStack*. Therefore instances of these two classes are provided as internals. Because the synchronization that is defined by these two classes needs to be combined, their synchronization filters are directly used for this class. This realizes an AND-condition for the constraints in the subsequent filters (since the constraints imposed by both filters have to be satisfied in order to let the message be accepted). The last filter dispatches messages to the respective internals; note that their filters have to be passed again, (redundantly) imposing the same synchronization constraints again.

```
class LockingStack interface
      internals
            superStack:SyncStack;
            locker:Locking;
      inputfilters
            < locker.lockSync;
              superStack.sync;
              disp:Dispatch={ superStack.*, locker.* }; >
end;
```

Figure 12. Implementation of a subclass of *Stack* named *LockingStack*.

5.1.2. Sorting Array

An array of *N* objects of class *SortCell* is connected in a linear chain to sort a list of *N* integers. First all the *N* elements of the list to be sorted are put into the left-most element of the array. Next, one by one, N numbers are retrieved from the left-most element of the array which outputs the elements of the input list in a sorted fashion. This example is structured in a fashion similar to the sorting array described by Brinch-Hansen [Brinch-Hansen 78].

```
class SortCell interface
comment objects of class SortCell are used to construct an array of N objects connected
            in a lineair chain to sort a list of N integers. ;
methods
      put(Integer) returns Nil;
      get returns Integer;
      connect(Pointer(SortCell)) returns Nil;
      update returns Nil;
conditions
      connected; empty; updateable; filled;
inputfilters      // default mutual exclusion cannot be used because the connect method stays active.
      < cellConnect : Error = {connect, connected => *.*};
            // only when the node is connected, other methods are allowed
        cellProtect : Error = { Protected=>update, *\update} };   // update is a protected method
        cellMutEx : Wait = {free=>*.*};
        cellSync : Wait = { connect, updateable=>update, empty=>put, filled=>{put, get} }; >
end;

class SortCell implementation
insvars
      next : Pointer(SortCell);
      items, value, temp, right : Integer;
      iAmConnected : Boolean;
conditions
      free returns (^self.active  - ^self.activeFor(update) ) = 0;
      connected returns iAmConnected ;
      empty returns ((items = 0) and (right < 1));
      filled returns (items = 1);
      updateable returns ((items=2) or ((right>0) and (items=0));
initial
      begin  items:=0; right:=0;   end;
```

```
methods
update
    begin
        if items=2
        then begin next.deref.put (temp); right := right + 1; end
        else begin value := next.deref.get; right := right - 1; end;
        items := 1;
    end;   // update
connect(neighbor :Pointer(SortCell) )
    begin
        next := neighbor;
        iAmConnected := true;
        return nil;   // early return: the method continues executing the following statements
        while true do
            self.update;
    end;   // connected
put(new : integer)
    begin
        items := items + 1;
        if items=2
        then if value > new
            then begin temp := value; value := new; end
            else temp := new;
        else value := new
    end;   // put
get
    begin items := 0;  return value;  end;
end;        // of class SortCell
```

Figure 13. Implementation of class *SortCell* and its use.

Each element of the array is an object of class *SortCell* shown in figure 13. This object has three interface methods *put*, *get* and *connect*. The method connect is used during initialization to give to each object the identity of the object on its right so that it can perform *put* and *get* operations on that object. This identity is stored in a local object called *next*, which is a pointer to an instance of class *SortCell*.

During the initialization phase, all instances of class *sortCell* are connected in a linear chain by giving each element the identity to its right neighbor using the interface method *connect*. Then the elements to be sorted are to be sent to the first sorting cell with the *put* method, and afterwards retrieved again (in sorted order) from the first sorting cell using the *get* method.

Initially every object has no item object stored in it. An object receives an item from its left neighbor, it keeps the smallest value seen so far with itself and forwards all larger values to the object on its right by invoking the *put* operation on that object. It keeps a count of the values sent to the right neighbor in an instance variable called *right*.

When an object receives a *get* request from its left neighbor, it forwards its own value to the left neighbor, and if *right* is greater than zero then it invokes the *get* operation on its right neighbor. An element of this array is in an equilibrium state either when *items* is 1, or when *items* is 0 with *right* also equal to 0. This equilibrium is disturbed when its left neighbor either removes an item or sends a new item; then the *update* method must be called.

5.1.3. Circulating Token System

A circulating token can be used to implement mutual exclusion of critical sections scattered over several objects in a distributed system. These objects are connected in a virtual ring configuration. There is one unique token in the system, and an object currently holding the

token executes the critical section operation and then passes the token to the object on its right. Concurrently executing objects are represented by instances of class *Node*. Each instance is given the identity of its right neighbor using the interface method *connect*. The *injectToken* method of an object is executed by its left neighbor to give it the token. The method *rotateToken* moves the token from the object to its neighbor node.

```
class Node interface
comment  this class defines a node in a circulating token system;
externals
       sharedData : Integer;
methods
       injectToken returns Nil;
       rotateToken returns Nil;
       connect (Pointer(Node)) returns Nil;
       criticalSection returns Nil;
conditions
       free; connected; hasToken;
inputfilters
       < nodeConnect : Error = {connect, connected=>Node.*};
         nodeMutEx : Wait = {free => *.*};
              // default mutual exclusion cannot be used since the initial method stays active.
         nodeSync : Wait = { hasToken => criticalSection, *.*\{criticalSection} }; >
end; // Node

class Node implementation
insvars
     next : Pointer(Node) ;
     iAmConnected, token : boolean;
conditions
     free returns (^self.active - ^self.activeFor(connect) ) = 0;
     connected returns iAmConnected ;
     hasToken returns token ;
methods
     connect(neighbor : Pointer(Node) )
          begin
               next := neighbor;
               iAmConnected := true;
               return nil;   // early return
               // now start executing the critical section repeatedly
               while true do
                    self.criticalSection;
          end;
     injectToken
          begin  token := true;   end;
     rotateToken
          begin  token := false;  next.deref.injectToken;   end;
     criticalSection
          begin
               // perform some operation on the shared data:
               sharedData := sharedData+1;
               self.rotateToken;
          end;   // criticalSection
end; // Node
```

Figure 14. Implementation of class *Node* which is an element of a circulating token system.

5.2. An Example of Parallel Computation

This is an example of an object that provides a facility to perform some function evaluation. This is a memory-less object in the sense that it does not maintain any permanent state information. Figure 15 shows an object that computes the factorial of an integer number. It has one interface method called *evaluate* which requires an integer parameter. An invocation

of this method returns the factorial of the parameter's value. Notice that there can be any number of client objects concurrently using this object.

```
class Factorial interface
    methods
        evaluate(Integer) returns Integer;
    inputfilters
        < mutex:Wait={*};           // override mutual exclusion; allow unlimited concurrent threads
          disp:Dispatch={*}; >      // straightforward dispatch
end;

class Factorial implementation
    methods
        evaluate(n:Integer)
            begin  if n=0 then return 1  else return server.evaluate(n-1)*n;  end;
end;  // of class Factorial implementation
```

Figure 15. Class Factorial as a concurrent calculator object.

An extension to class *Factorial* is implemented by a subclass *BoundedFactorial* which limits the maximum number of the concurrent executions. This maximum value is given to the object by invoking the method *limit* with the limiting value as an argument. If there are more requests than the allowed value, then these requests are put into a queue.

```
class BoundedFactorial interface
    internals
        fac : Factorial;
    methods
        limit(Integer) returns Nil;
    conditions
        belowLimit;           // is it allowed to create a new thread?
        ^self.isRecursive;    // declare the condition 'isRecursive' as defined by the object manager.
    inputfilters
        < recursiveFirst:Wait={inner.*, isRecursive=>*, belowLimit=>*};
          defMutEx:Wait={ True=>inner.*, belowLimit=>* };
          disp:Dispatch={fac.*, inner.*}; >        // inherit from Factorial, add new method(s)
end;

class BoundedFactorial implementation
    instvars
        max : Integer;   // the maximum no. of threads
    conditions
        belowLimit
            begin
                return (^self.dispatched-^self.completed-^self.waiting)<max
            end;
    methods
        limit(n:Integer)
            begin max:=n;  end;
end;  // of class BoundedFactorial implementation
```

Figure 16. Class *BoundedFactorial* is an object that limits the amount of internal concurrency.

The condition *belowLimit* is valid when the number of actively executing processes (i.e. non-blocked processes) is less than *max*, which is the current limit. In order to give priority to recursive calls, the first filter, *recursiveFirst*, accepts recursive messages even when there are no free threads. The second filter takes care that the recursive message will be blocked anyway until there is room for another thread.

Note that special precaution has to be taken since it is possible to change the limit on threads to a number that is lower than the number of threads active at that time. In this case, the

recursive calls receive precedence over the newly received *evaluate* messages. An optimization might be to use a priority-queue, in order to finish the threads that are almost ready (thus having a small value for the parameter) first.

5.3. Examples of Scheduling Problems

This section presents some examples where the execution order of the requests is required to be (re-) arranged to satisfy certain scheduling policies. In general, the language constructs to implement server objects and to handle request messages within such an object should allow the server to process the request messages and to respond to them in an order that may be different from the order of arrival of the request messages.

5.3.1. Priority Queue

The class *PriorityQueue* that is shown in figure 17 demonstrates two features: the first feature is synchronization based on message content, the second feature is the fact that the execution order of the received messages can be different from the reception order. The class *PriorityQueue* offers a method *prioMsg* on its interface, which takes an integer argument indicating the priority, and a block argument containing the operations which are to be performed:

```
class PriorityQueue interface
    comment  this class defines a priority queue. a message can be added to the queue by sending a prioMsg;
    methods
        prioMsg(integer, Block) returns Any;
        prioMsg2(integer, Block) returns Any;
        execOne returns Nil;
    conditions
        Internal;
        allowExecOne;
        allowPrioMsg2;
    inputfilters
        < protected : Error = { Protected=>{prioMsg2, execOne}, prioMsg };
            // external clients can only send prioMsg
          sync : Wait = { prioMsg, allowExecOne=>execOne, allowPrioMsg2=>prioMsg2 }; >
end; // PriorityQueue

class PriorityQueue implementation
    insvars
        prio : integer;
        sorter : SortedList;
    conditions
        allowExecOne       // there must be a waiting prioMsg2
            begin  return ^self.blockedFor(prioMsg2);  end
         allowPrioMsg2       // no prioMsg2 executing currently & priority match
            begin  return (^self.activeFor(prioMsg2)=0) and  (message.arg(1)=prio);  end;
    methods
        prioMsg(prio:Integer, b:Block)
            begin  sorter.put(prio);  return self.prioMsg2(prio, b);   end;
        prioMsg2(prio:Integer; b:Block)
            begin  return b.value; self.execOne;  end;
        execOne
            begin  prio:=sorter.get;  end;
end;         // PriorityQueue alternative implementation
```

Figure 17. Class *PriorityQueue.*

The approach that is followed is to execute the message *prioMsg* as soon as possible, and reschedule it by sending a message *prioMsg2* to the priorityqueue object again, while keeping

an administration of the priorities of the rescheduled messages in a sorted list. The method *execOne*, of which subsequent executions are mutually exclusive, picks the highest priority from the sorted list and executes the *first* message *prioMsg2* in the queue with that priority.

5.3.2. Alarm Clock

This example demonstrates how to implement the blocking of a message until a certain constraint is satisfied. Because this is analogous to the way Wait filters and conditions are functioning, this is straightforward to implement:

```
class AlarmClock interface
      methods
            wakeMeAt(Integer) returns Nil;
            tick returns Nil;
      conditions
            WakeUp;
      inputfilters
            < sleep:Wait= { True=>tick, WakeUp=>wakeMeAt };
              disp:Dispatch={*}; >
end;

class AlarmClock implementation
      instvars
            now:Integer;   // the current time
      conditions
            WakeUp
                  begin   if message.sel='wakeMeAt'  then return (message.args(1)=now)  else return false   end;
      methods
            wakeMeAt   begin end;  // nothing to do..
            tick   begin  now:=now+1;  end;
end;  // of class AlarmClock implementation
```

Figure 18. Class *AlarmClock*.

The class *AlarmClock* provides two methods, *wakeMeAt* and *tick*. The *tick* method is to be called every time a single time unit has passed, and updates the internal time of the *AlarmClock*, which is maintained by the instance variable *now*. The method *wakeMeAt* will only terminate when the time indicated by the argument becomes larger than the value of the variable *now*.

5.4. Examples of Resource Management Problems

5.4.1. Reader-Writer Synchronization Based on Mutual Exclusion for Write Operations

Reader/writer synchronization assumes the partitioning of the operations that access a resource into two kinds: *read* operations, which only retrieve information, and do not change the resource, and *write* operations, which do affect the state of the resource. In order to maintain consistency, write operations have to be performed with mutual exclusion -no other read or write method may be active simultaneously. However, it is possible to execute several read operations concurrently, which obviously may increase the throughput of the system.

We present two examples of classes implementing reader/writer synchronization; the first one, which is shown in figure 19, is a straightforward implementation, which only enforces

mutual exclusion during write operations. The condition *free* ensures that no method is executing within the object at all, which is a sufficient condition for allowing a write operation to continue. In order to execute a read method, it is required that there is no write method currently executing, which is ensured by condition *noWriter*.

```
class Reader interface
      externals
            resource : Any;
      conditions
            Nowriter;
      inputfilters
            < defMutEx:Wait={True=>* };              // all messages are always accepted
              deleg:Dispatch={resource.read}; >
end;

class Writer interface
      externals
            resource : Any;
      conditions
            NoWriter;
      inputfilters
            < writeSync:Wait={ NoWriter=>* };
              deleg:Dispatch={resource.write}; >
end;
class Writer implementation
      conditions
            NoWriter  begin return ^server.activeFor(write)=0;  end;
end;

class RdrWrtr interface
      comment this class implements RW, with reader priority;
      internals
            rdr:Reader;
            wrtr:Writer;
            resource:Any;    // could also be defined as an external
      inputfilters
            < defMutEx:Wait={ NoReader=>write, True=>read };
              rdr.readSync;
              wrtr.writeSync;
              deleg:Dispatch={ wrtr.write, rdr.read }; >
end;
```

Figure 19. Reader/Writer synchronization with reader priority.

This implementation however, gives priority to read methods; as long as one or more read methods are executing, newly arriving read messages are always allowed to execute. This might lead to starvation of write methods, in case new read messages keep on arriving. It is also possible to give priority to write methods, by allowing read methods only to execute when there are no write methods active or waiting or in the queue. But this approach again might lead to starvation of read methods.

An implementation that gives equal priority to readers and writers is provided in figure 20. Equal priority means that all messages are served on a First Come, First Serve (FCFS) basis, where subsequent read operations can execute in parallel. Such an implementation, however, cannot be realized by an object with a single message queue, since messages do not have any information regarding their position in the queue.

The implementation in figure 20 applies an internal object *rw* of class *RdrWrtr*, which implements the read and write methods, and enforces the constraints for maintaining consistency. The class *RdrWrtr_EP* dispatches messages to the *rw* object in FCFS order, but

only when no method is blocked in the queue of *rw*. This ensures that there will never be both read and write operations in the queue of *rw*, thereby making the reader-priority ineffective.

```
class RdrWrtr_EP interface
      comment This classes allows concurrent read operations, but
              enforces mutual exclusion for write operations. Read and
              write operations have equal priority: they are served on
              FCFS basis;
      internals
            rw : RdrWrtr;
      conditions
            NoWriterActive  ;   // no methods currently active within rw
      inputfilters
            < defMutEx:Wait = { NoWriterActive=>* };    // override default filter for mutual exclusion defMutEx,
              disp:Dispatch = { rw.* }; >    //  inherits all methods from RW_rp
end;

class RdrWrtr_EP implemenation
      conditions
            NoWriterActive  begin  return ^server.activeFor(write)=0  end;
end;
```

Figure 20. Reader/writer synchronization with equal priority.

This solution is only possible due to the composition of objects in case of inheritance; when a conventional class merging mechanism would be used, this would not introduce the additional queue which is used now.

6. Discussion

In section 3.4 we discussed the requirements for constructing concurrent systems. We will now analyze the composition-filters approach with respect to these requirements.

modular concurrency

Objects are the sole building blocks within the system, and thus modularity is ensured and supported on all levels (section 4.1). Each object is a potentially active entity because it can start its own, independent thread of activity through the initial method (section 4.1). Active objects can be nested (section 4.3). Synchronization within the system occurs on object level only; every object is responsible for its own synchronization and internal consistency (section 4.3.2).

expression-power

The synchronization mechanism of the composition-filters approach aims at a generic solution for synchronization problems. It was illustrated in section 5 through a number of examples in four categories that the mechanism is indeed capable of providing solutions for a wide range of synchronization problems. The inheritance hierarchies of the examples are shown in figure 21, illustrating how synchronization mechanisms can be both expressive and extensible. Class *SortCell* in figure 13 is an example for a parallel algorithm and employs concurrency both within and between objects. An implementation of a distributed algorithm is given by Class *Node* in figure 14. Class *RdrWrtr* in figure 20 implements equal priority for read-write synchronization, which depends on history information, i.e. the arrival order of

messages. Synchronization based on message content is exemplified by classes *PriorityQueue* and *AlarmClock* shown in figures 17 and 18, respectively.

The propositions for expressing synchronization constraints can be implemented by arbitrary message expressions, and thus have the power of message passing semantics.

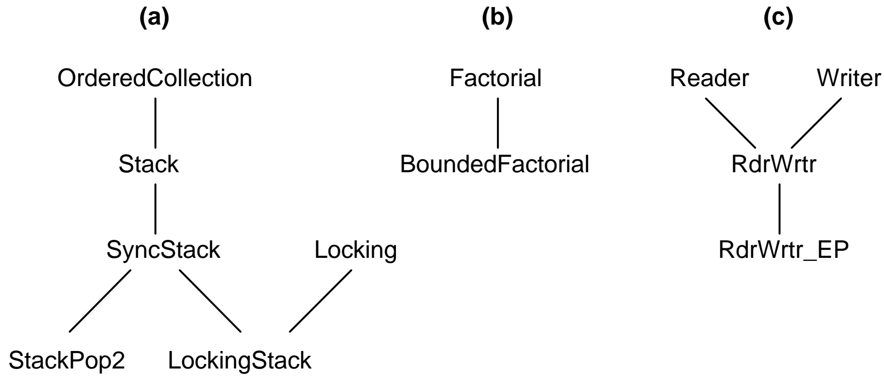


Figure 21. An overview of the inheritance hierarchies in our example applications: (a) Stack classes, (b) Factorial classes, (c) Readers/Writers synchronization

Internal concurrency

In Sina concurrent threads can be created by the use of the return statement or the initialization method. Sina objects do not enforce mutual exclusion unless this is expressed by filters (section 4.3.5) thereby allowing concurrent execution of received messages within a single object. Definition of *Factorial* and *BoundedFactorial* in section 5.2. illustrate how internal concurrency can be expressed and extended. In addition, Sina objects can encapsulate other objects that can be active themselves; this creates internal concurrency as well. The latter is applied in class *RdrWrtr* in figure 19.

Data-consistency

Data consistency within an object can be enforced by synchronizing using filters. To reduce the programming effort, the Sina compiler provides a set of default filters (section 4.3.5) enforcing mutual exclusion. Data consistency among objects is guaranteed by atomic delegations which is integrated with the filter mechanism. Atomic delegations are explained in another paper [Aksit 91].

Free from conflicts between inheritance and synchronization

I. Mixing Synchronization Code with Application Code

In Sina, the bodies of the methods are completely free from synchronization code. The conditions determining the synchronization constraints are programmed in condition implementations.

II. Key specifications

Filters define synchronization by associating conditions with methods. These methods can be referred to implicitly by using the wildcard symbol '*'. Because the filter specifications are completely independent from the method and condition implementations, changes to the class will have only a local affect on conditions and methods. This is illustrated by *Pop2Stack* in figure 10.

III. Non (de-)composable Synchronization Specifications

(a) In Sina synchronization specifications are decomposable. Firstly, they can be specified for every method separately. Secondly, the various conditions are implemented in separated conditions. Lastly, the synchronization specification of a single object may be divided over several filters. Examples are *LockingStack* (figure 12) and *BoundedFactorial* (figure 16).

(b) In Sina synchronization specifications are composable. Firstly, it is possible to write a new filter that combines the conditions and methods from more than one class, such as class *RdrWrtr* in figure 19. But it is also possible to directly combine filters from other ('super-') classes as illustrated by *LockingStack* in figure 12.

Concluding, we can state that the composition filters satisfy the requirements from section 3.4.

7. Conclusion

In this paper we presented the composition filters as a mechanism for specifying synchronization and concurrency control on objects. However, synchronization is only one aspect that is covered by composition filters. Other features, such as inheritance, delegation, multiple views, atomic transactions associative access and coordinated behavior, are realized through composition filters as well (see e.g. [Aksit 91, 92a, 92b]). The integration of all these features within a single mechanism avoids interference when more than one of these features are used for the same object. We consider this orthogonality to be a major contribution of the composition filters' synchronization mechanism.

With respect to the synchronization and concurrency control aspects of the composition filters approach, we can state that they satisfy all the requirements that were stated in section 3.4 for object-oriented concurrent programming languages (as was discussed in section 6.). More specifically, the composition filters do not suffer from the inheritance anomalies that we identified in sections 3.2 and 3.3., whereas most of the current object-oriented concurrent programming languages suffer from at least one these anomalies.

By solving a number of problems in several categories in section 5., we illustrated the expressive power of expressing synchronization and concurrency control with composition filters: e.g. intra-object concurrency, priority scheduling, synchronization based on message content. The inheritance relations between these examples show that it is possible to reuse and extend concurrent classes; both the synchronization code and the application code can be extended independently.

A prototype of a compiler for the Sina language, supporting composition filters, has been implemented and tested on Sun SPARC workstations. The compiler translates Sina code to C++ code. The implementation uses the AT&T C++ Task library to implement light-weight threads.

A number of issues were not covered in this paper; these are the subject of our current and future research. One of these issues is the possible combination of synchronization filters with other types of filters. For example, using both atomic transactions and (explicit)

concurrency control for a single object, or using both associative access or multiple views in combination with synchronization filters. Another issue is the dynamic configuration of objects, which allows for replacing conditions or interface objects at run-time. This might be used to realize dynamic scheduling or concurrency control policies (in [Tomlinson 89] and [Shibayama 91] similar dynamicity of synchronization policies is discussed). A final aspect is the relation of synchronization filters with the so-called *Abstract Communication Types* ([Aksit 89], [Aksit 92b]), which allow for abstracting the message flows between objects. Changing the synchronization properties of message flows, and the synchronization of coordinated behavior are important utilizations of abstract communication types.

With the provision of a flexible and expressive mechanism for realizing extensible and reusable synchronization code, we had two goals in mind. The first is the provision of an object model that is sufficiently powerful for modeling large and complex concurrent applications. The second goal is to support the construction of large libraries of synchronization- and concurrency control code. The composition filters approach provides a synchronization mechanism that supports these goals.

Acknowledgements

Anand Tripathi has been involved in the development of the synchronization scheme for the Sina language. We are also largely indebted to Satoshi Matsuoka, whose work on inheritance anomalies has been an important influence for us, and with whom we had fruitful discussions. We would also like to thank William van der Sterren, who implemented many examples. Charles Grossman provided us with many useful editorial comments.

References

[Ada 80] Honeywell, "Reference Manual for the Ada Programming Language", Minneapolis, Minnesota, 1980

[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*, OOPSLA '88, pp. 265-275, also: Memorandum INF-88-11, Department of Computer Science, University of Twente, the Netherlands, 1988

[Aksit 89] M. Aksit, "Abstract Communication Types", in "On the Design of the Object-Oriented Language Sina", PhD dissertation, Chapter 4, 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, K. Wakita, J. Bosch, L. Bergmans & A. Yonezawa, "Abstracting Inter-Object Communications Using Composition-Filters.", Memoranda Informatica 92-??, University of Twente, 1992

[Almes 85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, Jerre D. Noe, "The Eden System: A Technical Review", IEEE Transactions on Software Engineering, SE-11 , No. 1, 1985, pp. 43-59

[America 87] P. America, "POOL-T: A Parallel Object-Oriented Language", in "Object-Oriented Concurrent Programming", eds. A. Yonezawa, M. Tokoro, The MIT Press, Cambridge, Mass. 1987, pp. 199-220

[America 90] P. America, "A Parallel Object-Oriented Language with Inheritance and Subtyping", OOPSLA '90, pp. 161-168, October 1990

[Andrews 81] Gregory R. Andrews, "Synchronizing Resources", ACM Transactions on Programming Languages and Systems, 3 , No. 4, 1981, pp. 405-430

[Andrews 83] Gregory R. Andrews, Fred B. Schneider, "Concepts and Notations for Concurrent Programming", Computing Surveys, 15 , No. 1, 1983, pp. 3-43

[Atkinson 77] R. Atkinson & C. Hewitt, "Synchronization in Actor Systems", ACM SIGART-SIGPLAN Symposium on Principles of Programming Languages, January 1977, pp. 267-280

[Atkinson 91] C. Atkinson, S. Goldsack, A. Di Maio & R. Bayan, "Object Oriented Concurrency and Distribution in DRAGOON", JOOP March/April 1991, pp. 11-18, 1991

[Birman 85] Kenneth P. Birman, Thomas A. Joseph, Thomas Raeuchle, Amr El Abbadi, "Implementing Fault-Tolerant Distributed Objects", IEEE Transactions on Software Engineering, SE-11 , No. 6, 1985, pp. 502-508

[Bjornerstedt 88] A. Bjornerstedt & S. Britts, "AVANCE: An Object Management System", OOPSLA '88, pp. 206-221

[Black 86] A. Black, N. Hutchinson, E. Jul & H. Levy, "Object Structure in the Emerald System", OOPSLA '86, pp. 78-86

[Bloom 79] Toby Bloom, "Evaluating Synchronization Mechanisms", Seventh International ACM Symposium on Operating System Principles, pp. 24-32, 1979

[Booch 90] G. Booch, "Object Oriented Design (with applications)", Benjamin/Cummings Publishing Company, Inc., 1990

[Bos 89] J. van den Bos & C. Laffra, "PROCOL; A Parallel Object Language with Protocols", OOPSLA '89, pp. 95-102

[Brinch-Hansen 75] Per Brinch Hansen, "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering, SE-1 , No. 2, 1975, pp. 199-207

[Brinch-Hansen 78] Per Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, 21 , No. 11, 1978, pp. 934-941

[Campbell 74] R.H. Campbell & A.N. Habermann, "The Specification of Process Synchronization by Path Expressions", LNCS, 16, Springer, Berlin, pp. 89-102, 1974

[Campbell 80] R.H. Campbell & R.B. Kolstad, "An Overview of Path Pascal's Design and Path Pascal User Manual", SIGPLAN Notices, 15, No. 9, pp. 13-24, September 1980

[Campbell 89] R.H. Campbell, G.M. Johnston, P.W. Madany & V.F. Russo, "Principles of Object-Oriented Operating System Design", University of Illinois at Urbana-Campaign, Technical Report No. TTR89-14, April 1989

[Caromel 89] D. Caromel, "Service, Asynchrony, and Wait-By-Necessity", JOOP November/December 1987, Vol. 2, No. 4, pp. 12-22

[Caromel 90] D. Caromel, "Concurrency: An Object-Oriented Approach", TOOLS-2, (eds.) J. Bezivin, B. Meyer & J.M. Nerson, pp. 183-197, 1990

[Coad 91a] P. Coad & E. Yourdon, "Object-Oriented Analysis", 2nd Edition, Yourdon Press, 1991

[Coad 91b] P. Coad & E. Yourdon, "Object-Oriented Design", Yourdon Press, 1991

[Coulouris 88] G.F. Coulouris & J. Dollimore, "Distributed Systems-Concepts and Design", Addison-Wesley, 1988

[Decouchant 89] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill & X. Rousset de Pina, "A Synchronization Mechanism for Typed Objects in a Distributed System", Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, SIGPLAN

[Dijkstra 68] E.W. Dijkstra, "Co-operating Sequential Processes", in Programming Languages, 1968, pp. 43-112

[Gerber 77] A.J. Gerber, "Process Synchronization by Counter Variables", ACM Operating Systems Review, Vol. 11(4), October 1977, pp. 6-17

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

[Grass 86] J.E. Grass & R.H. Campbell, *Mediators: A Synchronization Mechanism*, Proceedings 6th International Conference on Distributed Computing Systems, Cambridge 1986, pp. 468-477

[Hoare 72] C.A.R. Hoare, "Towards a Theory of Parallel Programming", in "Operating Systems Techniques", (eds.) C.A.R. Hoare & R.H. Perrot, Academic Press, pp. 61-71, 1972

[Hoare 74] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", Communications of the ACM, 17 , No. 10, 1974, pp. 549-557

[Hoare 78] C.A.R. Hoare, "Communicating Sequential Processes", Communications of the ACM, 21 , No. 8, 1978, pp. 666-677

[Holt 83] R.C. Holt, "Concurrent Euclid, The Unix and Tunis", Addison-Wesley, 1983

[Kafura 89] D.G. Kafura & K.H. Lee, "Inheritance in Actor Based Concurrent Object-Oriented Languages", ECOOP '89, pp. 131-145

[Kafura 90] D. Kafura & K.H. Lee, "ACT++: Building a Concurrent C++ with Actors", JOOP May/June 1990, Vol. 3, No. 1, pp. 25-37

[Kallstrom 88] M. Kallstrom, Shreekant S. Thakkar, "Programming Three Parallel Computers", IEEE Software, January 1988, pp. 11-22

[Lieberman 86] H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior", OOPSLA '86, pp. 214-223

[Lieberman 87] H. Lieberman, "Concurrent Object-Oriented Programming in Act-1", in "Object-Oriented Concurrent Programming", (eds.) A. Yonezawa & M. Tokoro, MIT Press, 1987

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

[Loehr 92] K.-P. Loehr, "Concurrency Annotations", OOPSLA '92, 1992

[Matsuoka 90] S. Matsuoka, K. Wakita & A. Yonezawa, "Synchronization Constraints with Inheritance: What is Not Possible- So What is?", Tokyo University, Internal Report, 1992

[Matsuoka 93] S. Matsuoka, K. Wakita & A. Yonezawa, "Inheritance Anomaly in Object-Oriented Concurrent Programming Languages", to appear in "Research Directions in Concurrent Object-Oriented Programming", (eds.) G. Agha, P. Wegner & A. Yonezawa, MIT Press, April 1993

[Meyer 88] B. Meyer, "Object-oriented Software Construction", Prentice Hall, 1988

[Moss 85] J.E.B. Moss, "Nested Transactions: An Approach to Reliable Computing", MIT Press, 1985

[Neusius 91] C. Neusius, "Synchronizing Actions", ECOOP '91, (ed.) Pierre America, Lecture Notes in Computer Science 512, Springer-Verlag, 1991

[Nierstrasz 87] O.M. Nierstrasz, "Active Objects in Hybrid", OOPSPLA '87, pp. 243-253

[Nierstrasz 91] O.M. Nierstrasz, "The Next 700 Concurrent Object-Oriented Languages", Object Composition, (ed.) D. Tsichritzis, Centre Universitaire D'Informatique, 1991

[Okamura 91] H. Okamura & M. Tokoro, "ConcurrentSmalltalk-90", TOOLS-3, pp. 231-244, 1991

[Papathomas 91b] M. Papathomas & O. Nierstrasz, "Supporting Software Reuse in Concurrent Object-Oriented Languages: Exploring the Language Design Space", in "Object Composition", (ed.) D. Tsichritzis, Centre Universitaire D'Informatique, 1991

[Reed 79] D.P. Reed & R.K. Kanodia, "Synchronization with Event Counters and Sequencers", Communications of the ACM, pp. 466-481, October 1987

[Robert 77] P. Robert & J.-P. Verjus, "Toward Autonomous Descriptions of Synchronization Modules", Information Processing 77, North Holland, 1977, pp. 981-986

[Rumbaugh 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy & W. Lorensen, "Object-Oriented Modeling and Design", Prentice Hall, 1991

[Schaffert 86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian & C. Wilpolt, "An Introduction to Trellis/Owl", OOPSLA '86, pp. 9-16

[Shibayama 91] E. Shibayama, "Reuse of Concurrent Object Descriptions", TOOLS-3, pp. 254-266, 1991

[Snyder 86] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages", OOPSLA '86, pp. 38-45

[Stroustrup 86] B. Stroustrup, "The C++ Programming Language", Addison-Wesley, 1986

[Tokoro 86] M. Tokoro & Y. Ishika, "Orient84/K: A Language within Multiple Paradigms in the Object Framework", 19th Hawaii Conf. on System Sciences, pp. 198-207, January 1986

[Tomlinson 89] C. Tomlinson & V. Singh, "Inheritance and Synchronization with Enabled-sets", OOPSLA '89, pp. 103-112

[Tripathi 88a] A. Tripathi & M. Aksit, "Communication, Scheduling and Resource Management in Sina", JOOP, Vol. 1, No. 4, November/December 1988, pp. 24-37

[Tripathi 88b] A. Tripathi, "An Overview of the Nexus Distributed Operating System Design", IEEE Transactions on Software Engineering, January, 1988

[Wegner 87] P. Wegner, "Dimensions of Object-Based Language Design", OOPSLA '87, pp. 168-182

[Wirth 77] N. Wirth, "Modula: A Language for Modular Programming", Software: Practice and Experience, pp. 3-35, 1977

[Yokote 87] Y. Yokote & M. Tokoro, "Concurrent Programming in Smalltalk", Object-Oriented Programming, (eds.) A. Yonezawa & M. Tokoro, MIT Press, 1987

[Yonezawa 86] Akinori Yonezawa, "AI Parallelism and Programming", INFORMATION PROCESSING 86, pp. 111-113

[Yonezawa 87] A. Yonezawa et al., "Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1", in "Object-Oriented Concurrent Programming", (eds.) A. Yonezawa & M. Tokoro, MIT Press, pp. 55-89, 1987

[Yonezawa 90] A. Yonezawa (ed.), "ABCL-- An Object-Oriented Concurrent System", MIT Press, 1990