Examples of Reusing Synchronization Code in Aspect-Oriented Programming using Composition-Filters

Invited paper

Mehmet Aksit
Lodewijk Bergmans
Department of Computer Science
University of Twente
P.O. Boz 217, 7500 AE
Enschede, The Netherlands
(aksit | bergmans)@cs.utwente.nl

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* or *crosscutting* aspects have been extensively studied in the literature. As a solution to the synchronization reuse problems, we present the composition-filters approach. Composition filters can express synchronization constraints and operations on objects as modular extensions. In this paper we briefly 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.

Examples of Reusing Synchronization Code in Aspect-Oriented Programming using Composition-Filters

Invited paper

Mehmet Aksit
Lodewijk Bergmans
Department of Computer Science
University of Twente, P.O. Box 217,
7500 AE, Enschede, The Netherlands
(aksit | bergmans)@cs.utwente.nl

1 Introduction

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* or *crosscutting* aspects have been extensively studied in the literature [9].

As a solution to the synchronization reuse problems, we present the composition-filters approach. Composition filters can express synchronization constraints and operations on objects as modular extensions. In addition, the composition-filters approach is able to express various different aspects in a reusable manner.

In this paper we briefly 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.

2 Composition-Filters

2.1 Definitions

The composition-filters approach aims to enhance the expression power and reusability of objects. Filters are based on the following principles:

1. There are a number of pre-defined filter classes, each responsible for expressing a certain aspect.
2. Instances of a filter class can be created and attached to a class defined in various languages such as Smalltalk and C++. Filter classes are referred to as filter classes or filters, and the later as language classes or classes. Some filters may demand certain features from the language environment such as concurrency and/or real-time scheduling.
3. An instance of a filter class can be defined and attached to a class by using the filter interface definition language. A minimal filter interface definition consists of a class name and an **inputfilters** clause[1]. In Figure 1, *SyncStack* is the class name, and *sync* and *disp* are instances of filter classes *Wait* and *Dispatch*, respectively:

```
class SyncStack interface
    inputfilters
        sync:Wait={NonEmpty=>pop, True=>*\pop };   // specifies synchronization constraints
        disp:Dispatch={ coll.* };                  // provides all the  methods of  OrderedCollection
end;
```

Figure 1. A minimal interface definition for *SyncStack*.

[1] There are also output filters. For simplicity we do not discuss these filters here. Interested readers may refer to [5].

4. A filter instance can be initialized using a filter expression. The following expression is used to initialize *sync*: "sync:Wait= {{NonEmpty=>pop, True=>*\pop};". The second filter expression "disp:Dispatch={ coll.* };" is used to initialize *disp*. These are declarative specifications in that they do not make any assumptions about how they can be implemented[2].
5. 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*. The expression "disp:Dispatch={ coll.* };" means that *disp* delegates all the received messages to object *coll*.
6. The synchronization and delegation operations are based on a filter message manipulation process. If a filter is attached to a class, and if an object is created from that class, then the attached filter may manipulate the messages received[3] by the object. A message manipulation operation may change the implicit attributes of the received message. The implicit attributes are typically the identities of the receiver and the sender objects, the name of the method to be invoked, and zero or more arguments. The language environment may add extra attributes to the message, such as real-time constraint values. In the expression "disp:Dispatch={ coll.* };", *coll* is an instance of class *OrderedCollection*. Here, the received message is manipulated by replacing the identity of the receiver with the identity of *coll*, and the *self-variable* with the identity of the current instance of class *SyncStack*. *Dispatch* implements a *true delegation* mechanism as defined by Liebermann [17]. This requires that the delegating object (here instance of *SyncStack*), must be always referable by the delegated object (here *coll* of *OrderedCollection*), through a pseudo variable such as *self*. To distinguish from the inheritance-based *self-reference*, we introduce a new pseudo variable called *server*, which refers to the delegating object. The detailed description of filter manipulation operations are given in [15].
7. Typical manipulation operations are matching and/or substituting. For example, if the condition *NonEmpty* is true, the first filter matches the message with a selector *pop*. If the condition is false, and/or selector is not *pop*, then this filter matches any selector except *pop*.
8. A filter specification may depend on the state of its object. For example, in the first filter specification, the condition *NonEmpty* is true if there are one or more elements in the stack.
9. A filter expression is composed of one or more filter elements. These elements can be combined using logical operators such as CONDITIONAL OR, CONDITIONAL AND, and EXCLUSION. Here, the character "," implements a CONDITIONAL OR operation, which means that if the expression on the left-hand-side cannot match, then the expression on the right-hand-side will be evaluated. The character "\" is an exclusion operation. A CONDITIONAL AND operation can be implemented by cascading filters, using the ";" sign in the filter definition language. For example, in Figure 1, the filters *Wait* and *Dispatch* are composed together in a CONDITIONAL AND manner.

---

2 A filter can be implemented in various ways, for example, as a run-time entity by using message reflection, or as an in-lined code, by using compilation techniques.

3 In case of output filters, messages sent from the object are manipulated.

10. A filter specification refers to the parameters of the received messages only. It does not make any assumption about other filters. A filter may, however, refer to the conditions of its object, which can be accessible through the interface operations of the object.
11. A filter expression may also refer to object's interface variables, and some other external variables. For example, the expression disp:Dispatch={ coll.* } refers to the interface variable *coll* of class *OrderedCollection*. This mechanism is used for behavior composition, such as delegation. If a message is delegated to an interface object (here *coll*), the encapsulating object (here instance of *SyncStack*), inherits the interface behavior of the interface object (here instance of *OrderedCollection*) through the delegation mechanism.
12. Filter classes adopt similar initialization syntax. They differ from each other in how they react to the manipulated messages. For example, when a message is accepted by an instance of filter 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 message is accepted by an instance of filter class *Dispatch*, the message is delegated to specified object. If, however, the evaluation is not successful, the message passes to the next filter.
13. For type checking purposes, the filter interface definition language may require additional declarations. Figure 2 shows an example of an extended interface specification.
14. Programmers may introduce new filters, provided they fulfil the conditions[4].

```
class SyncStack interface
    comment inherits from class OrderredCollection, 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
        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
    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={ coll.* };                  // provides all inherited methods from OrderedCollection
end;
```

Figure 2. An extended interface definition for *SyncStack*.

In Figure 2, the interface object *coll*, the interface methods *isEmpty*, *pop* and *push*, and the filter condition *NonEmpty* of *SyncStack* are declared. This allows type checking between the filter interface specification and class *SyncStack*. Note that class *SyncStack* inherits from *OrderredCollection*, making all the methods of class *OrderedCollection* available at the interface of *SyncStack*. Further, class *SyncStack* introduces synchronization constraints.

2.2 Assumptions about the Language Classes

In principle the composition filters approach can be used for different object-oriented languages. However, filters generally depend on conditions, which are methods implemented

---

4 In our current implementation of Sina [16], implementing a new filter class requires sub-classing the language class *Filter* and overriding several operations. The filter compiler recognizes the newly introduced filter classes automatically.

in some language. Some filters, such as *Wait*, require certain expression power from conditions. A filter depends on the names of conditions only, but not on how conditions are implemented. To be able to give concrete examples, we made some assumptions about the implementation of the conditions used in this paper.

We assume that every object has a manager, which is responsible for receiving, buffering and dispatching messages. It therefore 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" as the target of a message invocation. Each object manager is encapsulated within its object, and cannot be accessed by other objects. Apart from monitoring the received messages, the object manager also provides support for returning results of message invocations.

The object manager also provides methods for retrieving the values of synchronization counters [14]. 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[5] 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 [13] and 'Synchronizing Actions' [18], and are especially useful for managing intra-object concurrency. As an example, we show how mutual exclusion can be defined by a filter:

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

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

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 the following: This filter is provided by default for all objects.

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

Finally, the language must provide means to create concurrent executions. We assume that the method *reply()* is provided by the object manager and returns its argument to the sender,

---

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

while the requested method may continue processing subsequent statements after the result has been returned.

The synchronization counters, and the method *reply* may be implemented in various ways. In our Smalltalk implementation, for example, we defined our own process scheduler by using the semaphore and process fork mechanisms of the Smalltalk language [16].

2.3 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 3 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 3. 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;
```

3 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. We also used the Sina language [9] to express the

implementation of the classes. In principle, any other language could be used provided that it supports the concurrency features explained in the previous section.

3.1 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.

3.1.1. Synchronization Extensions of Stack

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

The first example, as shown in Figure 4, gives the definition of class *Pop2Stack*, which inherits from class *SyncStack* 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 4. 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 5. 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 5. 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 a CONDITIONAL 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 6. Interface of class *LockingStack*.

3.1.1 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 [11].

```
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 7. Implementation of class *SortCell* and its use.

Each element of the array is an object of class *SortCell* shown in Figure 7. 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.

3.1.2 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 8. Implementation of class *Node* which is an element of a circulating token system.

3.2 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 9 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. This example illustrates the use of the pseudo variable *server*.

In Figure 10, an extension to class *Factorial* is implemented by its 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.

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.

```
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 9. Class Factorial as a concurrent calculator object.

```
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 10. Class *BoundedFactorial* is an object that limits the amount of internal concurrency.

3.3 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.

3.3.1 Priority Queue

The class *PriorityQueue* that is shown in Figure 11 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. 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.

```
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 11. Class *PriorityQueue*.

3.3.2 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.

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*.

```
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 12. Class *AlarmClock*.

3.4 Examples of Resource Management Problems

3.4.1 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 13, 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 13. 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 14. 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 14 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 14. 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.

4 Conclusion

The synchronization mechanism of the composition-filters approach aims at a generic solution for synchronization problems. It was illustrated in section 3 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 15, illustrating how synchronization mechanisms can be both expressive and extensible. Class *SortCell* in Figure 7 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 8. Class *RdrWrtr* in Figure 14 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 Figure 11 and Figure 12, respectively.

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

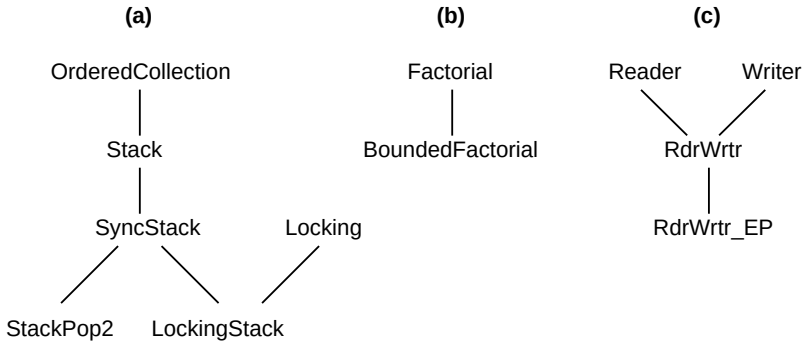

Figure 15. An overview of the inheritance hierarchies in our example applications:
(a) Stack classes, (b) Factorial classes, (c) Readers/Writers synchronization

The Composition-Filters approach is capable of expressing various different kinds of aspects in a uniform manner. In this paper, only the synchronization aspect has been described due to the space limitations. Each filter provides extensibility within its aspect domain. For example, real-time filters provide reusable real-time specification [6]. In addition, each aspect

expressed by a filter can be composed easily with other aspects. The following list gives a list of aspects and filters published in the literature. Inheritance, delegation [1], atomic delegation [2], multiple views, dynamic inheritance, and queries on objects [3], coordinated behavior and constraints [5], real-time and Synchronization [6][10], distributed synchronization [7], and client-server architectures [12].

Several implementations have been written for the composition filters model. The language Sina directly adopts the CF model. This language has been implemented and integrated within the Smalltalk environment. Detailed information about the Sina compiler can be found in [16]. Extensions to CORBA is described in [12]. The modeling problems addressed by the composition filters have been described in various publications [4][8].

References

[1] M. Aksit and A. Tripathi, Data Abstraction Mechanisms in Sina/ST, Proceedings OOPSLA '88, ACM SIGPLAN Notices, Vol. 23, No. 11, pp. 265-275, November 1988.

[2] M. Aksit, J.W. Dijkstra and A. Tripathi, Atomic Delegation: Object-oriented Transactions, IEEE Software, Vol. 8, No. 2, March 1991.

[3] M. Aksit, L. Bergmans and S. Vural, An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach, ECOOP '92, LNCS 615, Springer-Verlag, pp. 372-395, 1992.

[4] M. Aksit and L. Bergmans, Obstacles in Object-Oriented Software Development, Proceedings OOPSLA '92, ACM SIGPPLAN Notices, Vol. 27, No. 10, pp. 341-358, October 1992.

[5] M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, Abstracting Object-Interactions Using Composition-Filters, In object-based distributed processing, R. Guerraoui, O. Nierstrasz and M. Riveill (eds), LNCS, Springer-Verlag, pp. 152-184, 1993.

[6] M. Aksit, J. Bosch, W. v.d. Sterren and L. Bergmans, Real-Time Specification Inheritance Anomalies and Real-Time Filters, ECOOP '94, LNCS 821, Springer Verlag, pp. 386-407, July 1994.

[7] M. Aksit and L. Bergmans, Composing Multiple-Client-Multiple-Server Synchronizations, in Proceedings of the IEEE Joint workshop on Parallel and Distributed Systems, pp. 269-282, April 1997.

[8] M. Aksit, B. Tekinerdogan, F. Marcelloni, & L. Bergmans. Deriving Object-Oriented Frameworks from Domain Knowledge. To be published as chapter in M. Fayad, D. Schmidt, R. Johnson (eds.), Object-Oriented Application Frameworks, Wiley, 1998.

[9] L. Bergmans, Composing Concurrent Objects, Ph.D. thesis, University of Twente, The Netherlands, 1994.

[10] L. Bergmans and M. Aksit, Composing Synchronisation and Real-Time Constraints, Journal of Parallel and Distributed Computing 36, pp. 32-52, 1996.

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

[12] A. Burggraaf, Solving Modelling Problems of CORBA Using Composition Filters, MSc. thesis, Dept. of Computer Science, University of Twente, 1997.

[13] 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

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

[15] P. Koopmans, Sina user's guide and reference manual, Dept. of Computer Science, University of Twente, 1995.

[16] P. Koopmans, On the design and realization of the Sina compiler, MSc. thesis, Dept. of Computer Science, University of Twente, 1995.

[17] H. Lieberman, Using Prototypical Objects to Implement Shared Behavior, OOPSLA '86, pp. 214-223, 1986.

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