Real-Time Specification Inheritance Anomalies and Real-Time Filters

Mehmet Aksit[1], Jan Bosch[1], William van der Sterren[2] and Lodewijk Bergmans[1]

[1]Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.
email: {aksit, bosch, bergmans}@cs.utwente.nl
ftp: ftp.cs.utwente.nl, directory: doc/pub/TRESE
www server: http://www_trese.cs.utwente.nl
[2]Department of Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven. e-mail: sterrenw@info.win.tue.nl

Abstract. Real-time programs are, in general, difficult to design and verify. The inheritance mechanism can be useful in reusing well-defined and verified real-time programs. In applications developed by current real-time object-oriented languages, however, changing application requirements or changing real-time specifications in sub-classes may require excessive redefinitions although this seems to be intuitively unnecessary. We refer to this as the *real-time specification inheritance* anomaly. This paper introduces three kinds of real-time specification inheritance anomalies that one may experience while constructing object-oriented programs. As a solution to these anomalies, the concept of *real-time composition filters* is introduced. Filters affect the real-time characteristics of messages that are received or sent by an object. Through proper configuration of filters, one can specify real-time constraints, and reuse of these constraints without causing inheritance anomalies.

1. Introduction

Object-oriented programming languages [9, 10] have gained popularity in non real-time applications. These languages are highly modular and provide protection through encapsulation. In addition, *classes* and *inheritance* enable programmers to reuse existing software.

Recently, there have been some attempts to define real-time object-oriented languages [7, 12, 13, 15, 18]. One motivation for introducing object-oriented concepts in real-time languages is to reduce the complexity of applications through modularization so that *predictability* and *reliability* of applications can be increased. Real-time programs are, in general, difficult to design and verify. The inheritance mechanism can be useful in reusing well-defined and verified real-time programs. In addition, since object-oriented languages are now frequently applied to software implementations, it would not be practical to adopt different languages for real-time and non real-time parts of an application.

There are, however, several issues to be addressed in order to fully utilize object-orientation in real-time application development. We think that the *application code* must be reused separately from its real-time specifications. This promotes the reuse of both application code and real-time specifications. If separation is not possible, changes made to the application requirements or real-time specifications in sub-classes may result in excessive redefinitions even though this seems to be intuitively unnecessary. We refer to this as the *real-time specification inheritance* anomaly. This paper introduces three kinds of real-time specification inheritance anomalies that one may experience while constructing object-oriented programs. These anomalies are explained by means of a set of examples.

As a possible solution to these anomalies, the concept of *real-time composition filters* is introduced. Filters affect the real-time characteristics of the received and sent messages. By proper configuration of filters, one can specify real-time constraints, and reuse of these constraints without causing inheritance anomalies.

The following section introduces the basic real-time specification techniques and gives a short overview of real-time aspects in object-oriented methods and languages. Section 3 defines the concept of real-time specification inheritance anomaly through a number of examples. Section 4 introduces real-time composition filters and explains how filters can help in solving the anomalies. Finally, section 5 gives conclusions.

2. Real-Time Specifications in Object-Oriented Languages

2.1. Definitions

A *real-time system* is a system in which the correctness of its output(s) depends not only on the logical computations carried out but also on the time the results are delivered. The built-in notion of time and how it is used in the system is the difference between real-time systems and non real-time systems. The so-called *hard timing* constraints define a time-bound for a process that must be fulfilled, otherwise the computed result is useless, or can be even harmful. The process is not allowed to execute outside the specified time-bound. *Soft timing* constraints define a time-bound for a process outside which the computed result is not useless but still has a (diminished) value. The process is allowed to continue outside the specified time-bound. In addition, tasks in real-time systems are classified into *periodical* and *aperiodical* tasks. Periodical tasks have to be executed every p time units. The deadline of the task need not be equal to the start of the next period. Aperiodical tasks are not executed periodically but in reaction of an occurrence to a certain event.

The *scheduler* retrieves knowledge about the various timing constraints, resource requests and/or priority specifications, and schedules the tasks in such an order that an optimal performance of the system is achieved.

In the literature, a number of requirements are defined for expressing real-time constraints [5, 8]. For example, for a task one may define the following timing constraints: when it should start, when it should finish, whether it is a periodical or

single invocation, how much computation time it may take, and how long the execution may take.

In this paper we restrict ourselves to timing constraint specifications in object-oriented approaches and do not address other issues such as *priorities*, *scheduling algorithms* and *schedulability analysis of the real-time behavior* of programs[1].

In the object-oriented model the basic concepts are *objects*, *classes*, *messages*, *methods*, and *inheritance* [19]. Methods correspond to a single thread of execution, and therefore are the smallest entity on which timing constraints may be imposed. In most real-time object-oriented languages, timing constraints are associated with methods in one of three ways: (1) with a method header declaration; (2) with a part of method's implementation (will be referred to as a code block); (3) with a single message send or a statement in method's implementation.

An important aspect in object-oriented real-time modeling is the inheritance of real-time constraints through the inheritance hierarchy. Ideally, an object-oriented language must provide sufficient real-time specifications as one may expect from a real-time language, and reuse of these specifications through the inheritance mechanism. In addition, to improve robustness, encapsulation of implementation details with respect to *client objects* and/or *inheriting clients* must also be ensured.

Apart from specification inheritance, most languages combine the real-time specification of *server objects* with the specifications of *client objects*. In this case, the most restricting real-time specification is selected as a real-time constraint for both client and server objects.

2.2. Object-Oriented Real-Time Languages

Only in the recent years, real-time object-oriented programming languages have emerged. In the following paragraphs, we will briefly discuss some significant real-time object-oriented languages. The evaluation of these languages will be presented in section 3. Appendix A presents the real-time characteristics of these languages.

The Maruti programing language (MPL) [15] is based on the C++ language [9] and designed for the Maruti distributed operating system. MPL provides a number of constructs to specify timing constraints. These constructs can be applied to message sends and code blocks only. Timing constraints of the client object are passed on[2] to the server object and to the methods called by it.

RTC++ [12] is an extension of the C++ language and is suitable for programming both soft and hard real-time applications. RTC++ is implemented on top of the real-time distributed operating system kernel ARTS. Timing constraints can be associated

---

1 A priority [16] is a static label associated with a execution thread indicating its relative importance. It is used by a scheduling algorithm to determine the order of execution threads. Schedulability analysis [11] is a technique to determine in advance whether a program will meet its deadlines.

2 In real-time languages, sometimes the term inheritance is used for this purpose as well. In this paper, we only use the term inheritance when we refer to class inheritance.

either with method headers or declared in the method body. Thus timing constraints may be visible in the method interface and be encapsulated in the object's implementation.

The FLEX language [13] is based on the C++ language and associates real-time constraints with code blocks. Any change in the execution state causes only those immediately dependent constraints to be checked, thus no propagation of the constraints is done. The constraints may be labeled and thus referred to by other constraints. *Precedence relations* can be specified using *constraint blocks* that refer to attributes of other constraints. Constraints may also contain boolean expressions which are treated as an assertion to be maintained throughout the block's lifetime.

RealTimeTalk [7] is based on the Smalltalk language [10] without the features of Smalltalk that impede the timing prediction, such as method lookup and garbage collection. RealTimeTalk has been targeted to provide frameworks for soft and hard real-time applications. In the framework, objects of a special class *Use-Case* encapsulate timing-critical tasks. The RealTimeTalk compiler generates C-code which, in turn, has to be compiled to a target system.

The DROL language [18] is based on the C++ language and aims at programming distributed real-time systems. Its eminent feature is that users can describe the semantics of message communications at a meta-level as a communication protocol by using *sending* and *receiving* primitives. Like RTC++, DROL is implemented on top of the ARTS-kernel. The concept behind DROL is to provide the best effort at the server, taking into account the timing constraints of a message, and to realize the least suffering at the client side by detecting timing errors and avoiding the propagation thereof. Timing constraints can be specified for both messages and methods. For each method of a (server) object, its worst-case execution time may be specified. Each message send may be a so-called *time polymorphic invocation*. From a set of alternatives the server chooses the method that meets the timing constraint. Periodic actions are specified using the *active* keyword, followed by the method declaration and parameters specifying the time bound of period and execution time. Because of the time polymorphic invocations, DROL provides flexible computations and graceful degradation. In DROL, timing constraints for message sends may only be declared in method bodies. Timing constraints for *message acceptance* specify the worst case time and may only be declared at the object interface.

3. Real-Time Constraint Specification Inheritance Anomalies

A *real-time specification inheritance anomaly* is a conflict between the inheritance mechanism and real-time specifications in an object-oriented programming language; the conflicting characteristics of these mechanisms restrain simultaneous use of inheritance and real-time constraints.

It is important to note that the real-time specification inheritance anomaly is not inherent with combining real-time specifications and inheritance. On the contrary, the anomaly is fully language dependent. The way a language implements timing constraints and inheritance can be the sole cause of real-time specification inheritance anomalies.

We found the following types of real-time inheritance anomalies: *mixing real-time specifications with application code*, *non-polymorphic real-time specifications* and *orthogonally restricting specifications*.

3.1. Mixing Real-Time Specifications with Application Code

Real-time specifications are mixed with application code if the real-time specifications are associated with a part of the implementation of a method. In this case, since real-time specifications cannot be separated from the method implementation, it is impossible to redefine the real-time specification nor the method implementation without redefining both.

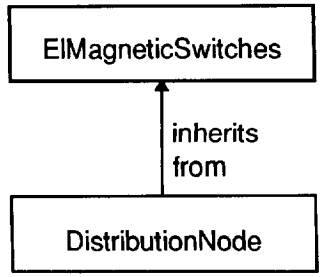
Consider for example, class *DistributionNode* which models a node in an electricity distribution network. This node has one production line and several consumption lines and it has to control the electricity flow from the production line to the consumption lines. The production line has a software controlled connection device that monitors the status of the line. If the production line is disconnected, then the node will disconnect all consumption lines within a predefined period of time.

Class *DistributionNode* can be seen as a specialization of an electro-magnetic switch. As listed in Figure 1, we therefore first define class *ElMagneticSwitches* with a set of methods that connects or disconnects a set of power lines. For each consumption line, class *ElMagneticSwitches* declares a pair of *connectLine* and *disconnectLine* methods. These methods control a set of relays and are declared between lines (3) to (6). The example classes are expressed using a general object-oriented notation.

```
(1) class ElMagneticSwitches interface
(2)     begin
(3)         methods
(4)             connectLine1 returns nil;        // connects power line 1 //
(5)             disconnectLine1 returns nil;     // disconnects power line 1 //
(6)             ..... // list of connectLine and disconnectLine methods for every remaining line //
(7)     end;
```

Fig. 1. Interface declaration of class *ElMagneticSwitches*.

Figure 2 shows the definition of class *DistributionNode*. Class *DistributionNode* inherits from *ElMagneticSwitches* and defines two additional methods called *productionLineDisconnected* and *disconnectConsumerLine*. The method *productionLineDisconnected* is invoked when the production line is disconnected for some reason. When this method is invoked, it disconnects all the consumer lines. The method *disconnectConsumerLine* accepts the line number as an argument, and disconnects the corresponding line.




```
(1) class DistributionNode interface
(2)     begin
(3)         inherits from ElMagneticSwitches;
(4)         methods
(5)             productionLineDisconnected returns nil;
(6)             disconnectConsumerLine(Integer) returns nil;
(7)     end;

(8) class DistributionNode implementation
(9) ..... // declarations and initializations //
(10)    begin
(11)        productionLineDisconnected
(12)            begin
(13)                for i:=1 to numberOfLines do timeMinEnd numberOfLines * t_consumer
(14)                    begin
(15)                        self.disconnectConsumerLine(i)        // self call //
(16)                    end;
(17)            end;
(18)        disconnectConsumerLine (lineNo: Integer)
(19)            begin
(20)                case lineNo of timeMinEnd t_consumer
                                        // real-time specification for disconnection //
(21)                    1: self.disconnectLine1;
                                        // inherits from ElMagneticSwitches //
(22)                    2: self.disconnectLine2;
                                        // inherits from ElMagneticSwitches //
(23)                        // cases for other lines //
(24)                end;
(25)            end;
(26)    end;
```

Fig. 2. Interface declaration of class *DistributionNode* and implementation of the methods *productionLineDisconnected* and *disconnectConsumerLine*.

In Figure 2, line (3) declares *DistributionNode* as a subclass of *ElMagneticSwitches*. Lines (4) to (6) declare the method headers. The implementation of the method *productionLineDisconnected* is given between lines (12) and (17). In line (15), this method calls the method *disconnectConsumerLine* for every existing consumer line. Following the keyword **timeMinEnd**, the amount of time available for the execution of the FOR-loop is specified as the number of consumer lines multiplied by the time required to disconnect a line. In lines (19) to (25), depending on the argument value, the method *disconnectConsumerLine* calls the method *disconnectLine* which is inherited from class *ElMagneticSwitches*. In line (20), the deadline for the method *disconnectedConsumerLine* is specified using the statement **timeMinEnd** $t_{consumer}$.

We will now illustrate the *mixing real-time specifications with application code anomaly* by introducing a subclass. As shown in Figure 3, assume now that class *FastDistributionNode* inherits from *DistributionNode*. The timing constraint for the methods *productionLineDisconnected* and *disconnectConsumerLine*, however, are more restrictive than the ones of class *DistributionNode*. In lines (5) and (6), the methods defined by *DistributionNode* are overridden by declaring them again in the subclass. This is necessary because the real-time specifications are embedded in the

implementation of the superclass, and therefore the entire method has to be redefined in the subclass. Note that in line (14), the real-time constraint is now specified as *numberOfLines* * $t_{consumerNew}$, where $t_{consumerNew} < t_{consumer}$.

```
(1) class FastDistributionNode interface
(2)     begin
(3)         inherits from DistributionNode;
(4)         methods
(5)             productionLineDisconnected returns nil;
                    // mixing specification with code anomaly //
(6)             disconnectConsumerLine(Integer) returns nil;
                    // mixing specification with code anomaly //
(7)     end;
(8) class FastDistributeNode implementation
(9) ..... // declarations and initializations //
(10)    begin
(11) ..... // declarations and initializations //
(12)        productionLineDisconnected
(13)            begin
(14)                for i:=1 to numberOfConsumerLines do
                                timeMinEnd numberOfLines * tconsumerNew
(15)                    begin
(16)                        self.disconnectConsumerLine(i)   // self call //
(17)                    end;
(18)            end;
(19) ..... // implementation of the method disconnectConsumerLine //
(20)    end;
```

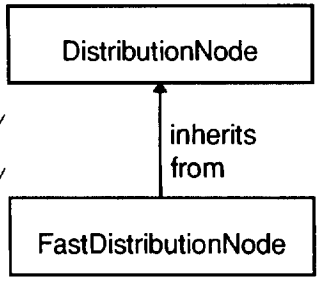


Fig. 3. Redefinition of the method *productionLineDisconnected* in the subclass *FastDistributionNode*.

To avoid this anomaly, one can associate real-time specifications with the method headers. For example, in lines (5) and (6) of Figure 4, the timing constraints of methods are associated with the method declarations. If the adopted language permits, the software engineer can then override the method header declarations only, without changing their implementations. We also think that declaring real-time specifications at the interface of an object fits more to the object-oriented programming style.

```
(1) class DistributionNode interface
(2)     begin
(3)         inherits from ElMagneticSwitches;
(4)         methods
(5)             productionLineDisconnected returns nil;
                    timeMinEnd numberOfLines * tconsumer; //associated with header //

(6)             disconnectConsumerLine(Integer) returns nil;
                    timeMinEnd tconsumer; // associated with header //
(7)     end;
```

Fig. 4. Interface declaration of class *DistributionNode* using constraint specification at the interface.

3.2. Non-polymorphic Real-Time Specifications

The *non-polymorphic real-time specifications anomaly* occurs if a real-time specification can not be *polymorphically* associated with a different method or a set of methods. The programmer is then forced to define real-time specifications for every method that requires the same specification. This anomaly can also be experienced if the implementation of a method has to be changed in the subclass without changing its real-time specification.

We will now give an example to illustrate this anomaly. Assume that we want to protect consumer lines against short circuits. As shown in Figure 5, we introduce a new class called *MultipleFuses* with a set of methods to detect short circuits on the corresponding lines. These methods are declared in lines (3) to (6). Since in case of a short circuit the corresponding line has to be disconnected within a certain time, all the method headers of *MultipleFuses* are associated with the real-time constraint specification using the statement **timeMinEnd** $t_{shortCircuit}$. As listed in lines (12) to (15), when a short circuit is detected, the corresponding method *disconnectConsumerLine* is invoked. Because the real-time specifications cannot be separated from the method header declarations and be polymorphically associated with all the corresponding methods, specifications had to be repeated for every method. Repetitive association of real-time specifications with method headers is obviously error-prone, especially if these methods are located in different classes of the inheritance hierarchy. A single change of the real-time constraints requires updating all the separately defined specifications. Moreover, explicit association restricts the open-endedness of the hierarchy because new methods with the same real-time specification cannot be introduced in the subclasses without explicitly associating the specifications with the new methods.

```
(1) class MultipleFuses interface
(2)     begin
(3)         methods
(4)             shortCircuitLine1 returns nil timeMinEnd t_shortCircuit;
(5)             shortCircuitLine2 returns nil timeMinEnd t_shortCircuit;
                                    // forced to declare the same spec. for all methods //
(6)             ..... // list of shortCircuitLine methods for every remaining line //
(7)     end;
(8) class MultipleFuses implementation
(9) ..... // declarations and initializations //.
(10)    begin
(11)        shortCircuitLine1
(12)            begin
(13)                self.disconnectLine1;   // deferred method (implemented by the subclass) //
(14)                .....                   // other operations, such as signaling, etc. //
(15)            end;
(16) ..... // implementation of other methods //.
(17)    end;
```

Fig. 5. Definition of class *MultipleFuses*.

3.3. Orthogonally Restricting Real-Time Specifications

If several real-time specifications are defined independently and combined through (multiple) inheritance and then affect each other semantically, the software engineer can be forced to redefine some of the related specifications. We call this problem the *orthogonally restricting real-time specifications* anomaly.

Consider for example, the interface definition of class *ProtectedDistributionNode* as listed in Figure 6. Class *ProtectedDistributionNode* combines the features of *MultipleFuses* and *DistributionNode*. Line (3) shows that this class inherits from classes *DistributionNode* and *MultipleFuses*. In case there is a short circuit on a consumer line, the corresponding method *shortCircuitLine* inherited from class *MultipleFuses* will be invoked. This method will disconnect the line by invoking the corresponding method *disconnectLine* which is inherited from class *ElMagneticSwitches*.

We assume that protection is only meaningful if the disconnection switches are at least as fast as the fuses. To ensure that the same real-time constraints are also valid for the disconnection methods, in lines (5) and (6), all the methods inherited from *ElMagneticSwitches* and used for disconnection are redefined. We observe here two kinds of anomalies. The *non-polymorphic specifications anomaly* is observed because the real-time specification used in class *MultipleFuses* has to be defined for every disconnection method repeatedly. The *orthogonally restricting specifications* anomaly occurs because, although the methods of classes *ElMagneticSwitches*, *DistributionNode* and *MultipleFuses* were defined separately, they semantically affect each other within class *ProtectedDistributionNode*.

In lines (7) and (8) the methods *productionLineDisconnected* and *disconnectConsumerLine* are redefined because their timing constraints are now affected by the redefinition of timing constraint $t_{shortCircuit}$ of the disconnection methods. Again, we experience the *orthogonally restricting specifications* anomaly[3].

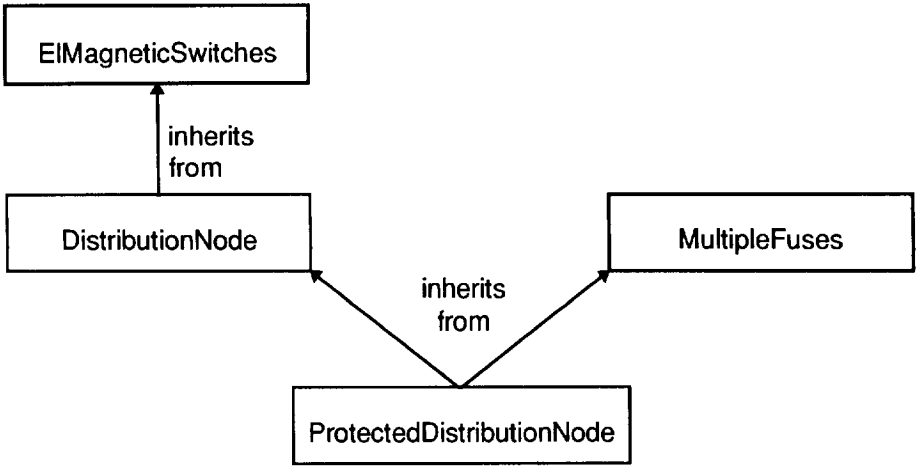
3.4. Other Anomalies

Often real-time applications consist of a number of tasks. Synchronization mechanisms are needed to control access to shared objects and to synchronize these tasks. In general, in addition to real-time specifications, synchronization constraints have to be inherited as well. Several researchers have mentioned that [6, 14] in concurrent object-oriented languages, introducing a new method and/or overriding

---

3 One may claim that the disconnection methods do not need to be redefined if the adopted language combines the real-time specification of the server object with the client object. We think, however, that the real-time specifications must be explicitly declared at the interface of objects. Secondly, this would not eliminate the anomaly for the methods *disconnectConsumerLine* and *productionLineDisconnected* since these methods are not called by the methods of class *MultipleFuses*, and therefore the real-time specification cannot be implicitly associated with them.

an inherited method in a subclass may require additional definitions. In this paper we only consider the real-time specification anomalies. For the synchronization anomalies in real-time languages we refer to [17].




```
(1) class ProtectedDistributionNode interface
(2)     begin
(3)         inherits from MultipleFuses, DistributionNode;
(4)         methods
(5)             disconnectLine1 returns nil timeMinEnd t_shortCircuit
                    // shortCircuitLine methods from MutipleFuses and from //
                    // ElMagneticSwitches are orthogonally restricting //

(6)             .....  // list of remaining disconnect methods with real-time specification for every line //

(7)             productionLineDisconnected returns nil;
                                timeMinEnd numberOfLines * t_shortCircuit;
(8)             disconnectConsumerLine(Integer) returns nil;
                                timeMinEnd t_shortCircuit;
                    // forced to redefine; methods from MultipleFuses and DistributionNode are //
                    // orthogonally restricting //
(9)     end;
```

Fig. 6. Redefinition of the methods disconnectLine, productionLineDisconnected and disconnectConsumerLine in the subclass ProtectedDistributionNode due to the orthogonally restricting specification anomaly.

3.5. Evaluation of Real-Time Object-Oriented Languages

We will now evaluate the real-time object-oriented languages with respect to the real-time specification anomalies.

The Maruti Programming language is only capable of expressing timing constraints within methods; timing constraints are always encapsulated and cannot be separately inherited nor accessed by subclasses. Thus, it suffers from all three anomalies we described.

In RTC++ timing constraints can either be declared at the method interface or declared in the method body. Programs written in RTC++ may suffer from all three real-time specification inheritance anomalies. RTC++ allows for timing specifications at method declaration level; thus the *mixing real-time specifications with application code* anomaly can be avoided. However, RTC++ does not prevent software engineers from mixing code and timing specifications. In addition, in RTC++ timing constraints cannot be separately defined, nor inherited.

In FLEX, timing constraints may only be associated with statements. It thus allows for mixing code with timing specifications. FLEX enables a software engineer to attach labels to blocks with constraints in order to refer to the start and finish time of the corresponding block. However, the constraints themselves cannot be referred to, so they cannot be inherited separately. Polymorphic constraints cannot be implemented as well.

RealTimeTalk does not suffer from inheritance anomalies because classes with real-time specifications cannot be subclassed.

In *DROL*, timing constraints for message sends may only be declared in method bodies. Timing constraints for message reception specify the worst case time and may only be declared at the object interface. DROL allows for mixing code with time polymorphic invocations, so it suffers from the *mixing real-time specifications with application code* anomaly.

4. An Approach to Solving Real-Time Specification Anomalies: Real-Time Filters

4.1. Real-Time Filters

To take advantage of the message-based nature of object-oriented languages, we propose to have timing information traveling with each message. This way, both client and server objects can impose timing constraints, and the server may take the client's timing constraint into account.

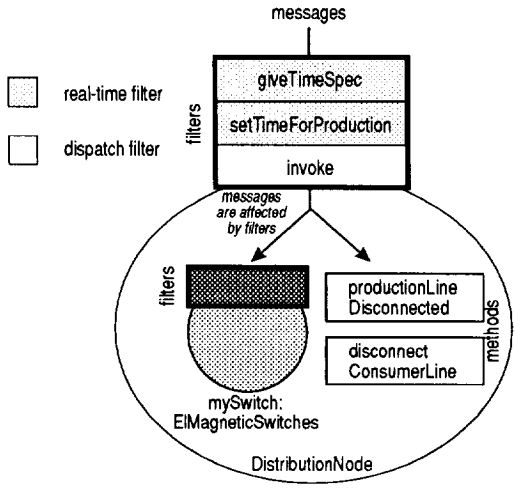
Since real-time constraints are being carried in messages, we need language constructs to read and affect timing constraints by client and/or server objects. We also believe that real-time specifications must be declared at the interface of objects. Obviously, we want to avoid real-time specification anomalies.

To fulfil these requirements, we extend the conventional object-model with the so-called *composition-filters*[4] . An example of using composition-filters is illustrated

---

4 We would like to emphasize that the composition-filters approach is a modeling paradigm rather than the definition of a language with fixed semantics. This is because the semantics of the programs expressed in this language are largely determined by the semantics of the adopted filters. For example, [1] illustrated how both inheritance and delegation can be simulated using filters. [2] introduced filters for defining reusable transactions. Language-database problems were addressed in [3]. [6] showed how composition-filters can be used to express reusable synchronization specifications. In [4],

in Figure 7. This figure shows the interface definition of class *DistributionNode*, whose functionality is the same as the one listed in Figure 4. We will compare the latter with the composition-filters implementation in Figure 7. Lines (3) and (4) declare internal object *mySwitch* of class *ElMagneticSwitches* at the interface of *DistributionNode*.




```
1) class DistributionNode interface
(2)     begin
(3)         internals
(4)             mySwitch:
ElMagneticSwitches;
(5)         methods
(6)             productionLineDisconnected returns nil;
(7)             disconnectConsumerLine(Integer) returns nil;
(8)         inputfilters
(9)             giveTimeSpec: RealTime = { True => self.* | time.minEnd( t_consumer ) | };
(10)            setTimeForProduction: RealTime = { True => self.productionLineDisconnected
(11)                                                | time.setEnd( numberOfLines *
                                                      time.getEnd ) | };
(12)            invoke: Dispatch = { True => mySwitch.*, True => inner.* };
(13)    end;
```

Fig. 7. Interface definition of class DistributionNode using real-time and dispatch filters.

Another difference is the declaration of three filters in lines (8) to (12) following the **inputfilters**[5] clause. A filter determines whether a particular message is either *accepted* or *rejected* and what action is to be performed in either case. Each filter is declared as an instance of a filter class. A programmer may define an arbitrary

---

filters were used to abstract coordinated behavior among objects. The application of composition-filters for real-time specifications was not published before.

5 In addition to input filters, the composition-filters model also supports output filters. Output filters affect outgoing messages, whereas inputfilters affect incoming messages. Output filters are useful for imposing real-time constraints on messages that are sent by an object. For brevity, we do not further consider output filters in this paper.

number of filters for an object. For example in figure 7, lines (9) to (11) declare two filters of class *RealTime*. Line (12) declares a filter of class *Dispatch*[6] .
A filter specifies conditions for message acceptance or rejection and determines the appropriate resulting action. In line (9), an input filter called *giveTimeSpec* of class *RealTime* is declared using the expression

$$\text{giveTimeSpec: RealTime = \{ True =>self.* | time.minEnd(} t_{consumer} \text{) | \};}$$

An input filter of class *RealTime* is used to affect the timing attribute of the message when the corresponding message matches with the filter. If the message does not match with the filter, then it will pass to the next filter without receiving the timing attribute. The filtering condition between the brackets "{" and "}" is specified as

$$\text{\{ True => self.* | time.minEnd(} t_{consumer} \text{) | \};}$$

On the left hand side of the characters "=>", a necessary condition is specified, denoted by the condition identifier, which is *True* 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 of external variables as well. Conditions are useful in controlling the object's interface, for example in assigning timing constraints based the urgency of the situation. In class *DistributionNode*, for simplicity, the constant *True* is used instead of a condition variable.

The received message is compared with the method names specified on the right hand side of the characters "=>". The character "*" indicates a wild-card or don't care condition; In case of *RealTime* filters, *self.** means that if the message matches with any of the method names provided at the interface of class *DistributionNode*, then it will receive the timing constraint expressed between the separators "| ... |". An alternative could be to list all the method names explicitly. The pseudo-variable *self* denotes the instance of class *DistributionNode*.

The real-time constraint in the filter is expressed as

$$\text{| time.minEnd(} t_{consumer} \text{) |}$$

---

6 The current version of the Sina language, the language that adopted the composition-filters model, provides a number of primitive filters such as *RealTime*, *Dispatch*, *Meta*, *Error* and *Wait*. The *RealTime* and *Dispatch* filters are explained in this section. The *Meta* filter is used to model coordinated behavior and explained in [4]. The *Error* filter is similar to the *Dispatch* filter but it does not provide a method dispatch; it raises an error condition if a message does not pass through the filter [3]. The *Wait* filter is used for synchronization [6]. These filters can be used as both input and output filters. An important feature of all these filters is that they are orthogonal to each other and, therefore, they can be combined freely.

Here, *time* denotes the timing attribute of the received message. The method *minEnd* is defined by *time* to associate a timing constraint with it. The method *minEnd* accepts a time specification as a value and assigns it as the deadline of the execution. However, this is only done if the new value is smaller than the previous value (the minimum is taken). In appendix B, the available methods of *time* are defined. The real-time filter that is defined in line (9), associates with every message that is received and accepted by this filter the timing value $t_{consumer}$, if this value is smaller that the current value.

In lines (10) and (11) the second real-time filter is declared using the expression

```
setTimeforProduction:RealTime =
{True=>self.productionLineDisconnected | time.setEnd( numberOfLines * time.getEnd) | };
```

The filtering condition here is *True* and therefore the received message will always be compared with *self.productionLineDisconnected*. If the message matches, then the timing specification between the bars "| ... |" will be considered for the message. Otherwise, the message will pass to the next filter.

The timing constraint in filter *setTimeforProduction* is defined as

```
| time.setEnd( numberOfLines * time.getEnd ) |
```

Here again, *time* denotes the timing attribute of the received message. The method *setEnd* is defined by *time* to assign a timing constraint to it. The method *setEnd* accepts a timing constraint as a value and assigns it as a timing attribute without considering the existing value. The method *getEnd* returns the current deadline of the received message. If the received message is invoked for the method *productionLineDisconnected*, then the real-time filter as defined in lines (10) and (11) will associate the timing value, which is *numberOfLines* times more than the previous value. Note that in this way, the timing constraints of both methods are always consistent with each other.

In line (12), the filter *invoke* of class *Dispatch* contains two filter elements. These two filter elements have the following meaning:

The first element of the filter, "*True=>mySwitch.**" specifies that all the incoming messages are delegated to the internal object *mySwitch*, provided that *mySwitch* (which is an instance of class *ElMagneticSwitches*) supports these messages. Since the methods of *ElMagneticSwitches* are now available to *DistributionNode*, class *DistributionNode* inherits the operations of class *ElMagneticSwitches*. This technique for simulating inheritance is also referred to as *delegation-based inheritance*. When an instance of class *DistributionNode* is created, its internal object *mySwitch* is also created. An important feature here is that instance variables of the *superclass* are only accessible through operations provided by the superclass. We would like to stress that internal objects differ from instance variables, because internals are used to compose the behavior of the object, whereas instance variables represent the local data of the object.

If the first filter element does not match with the message, the second filter element is evaluated. Instead of delegating to an internal object such as *mySwitch*,

this filter element delegates the message to the pseudo-variable *inner*[7] . By declaring *inner* as a target object, class *DistributionNode* makes the methods declared and implemented by itself available to its clients.

When a message with a timing attribute is dispatched to a method, the scheduler schedules the execution of the method based on the timing attributes of the message[8] .

4.2. Eliminating the Mixing Real-Time Specifications with Application Code Anomaly

We will now illustrate how real-time filters can avoid the *mixing real-time specifications with application code* anomaly. We redefine class *FastDistributionNode* as shown in Figure 8 and compare it with the previous implementation as listed in Figure 3. To implement inheritance, in lines (3) and (4) internal object *myNode* is declared as an instance of *DistributionNode*. In lines (5) to (7) two filters are declared. Here, *fastTimeSpec* is a real-time filter and *invoke* is a dispatch filter. The real-time filter *fastTimeSpec* has the following purpose: if the message matches with any of the method names provided at the interface of class *FastDistributionNode*, and if the timing value $t_{consumerNew}$ is smaller than the timing value of the message, then the message will accept the timing value $t_{consumerNew}$ and pass to the next filter. Otherwise, the message will pass to the second filter without receiving the timing value.

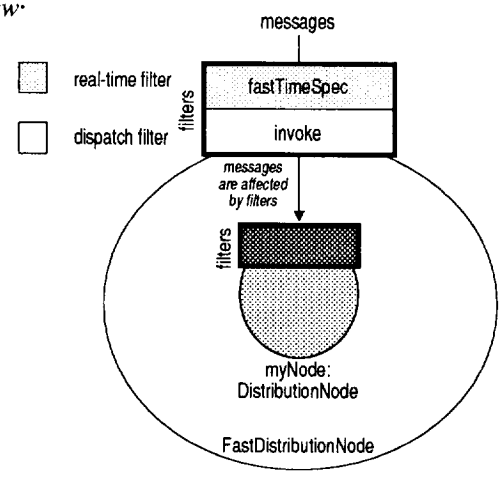
The dispatch filter *invoke* has a single filter element. The element *myNode.** specifies that all the incoming messages are delegated to the internal object *myNode*, provided that these messages are supported by class *DistributionNode*. Since the methods of *DistributionNode* are now available to *FastDistributionNode*, class *FastDistributionNode* inherits the methods of *DistributionNode*.

---

7 Apart from the pseudo-variable *inner*, two other pseudo-variables, *self* and *server*, are also available as means of self-reference. The variable *inner* allows direct internal access on the objects' own methods. *self* refers to the instance of the class which defines the method. If, for example, *class ElMagneticSwitches* refers to *self*, it will refer to *mySwitch* but not the instance of *DistributionNode*. We introduced *inner* to avoid infinitely nested compositions. Such nested compositions can be created if only *self* is used. When refering to the object that originally received the message, *server* is used as a target. For example, if *ElMagneticSwitches* refers to *server*, it will refer to the instance of *DistributionNode*. Note that *server* is dynamically bound and is equivalent to Smalltalk *self*.

8 Currently, we are experimenting with different scheduling algorithms such as *earliest dead-line first* [16].

Clearly, the composition-filters implementation of *FastDistributionNode* does not suffer from the *mixing realtime specifications with application code* anomaly. This is because the real-time specification is expressed through the use of a real-time filter which is defined at the interface of class *FastDistributionNode*. If the received message matches with the real-time filter, then the specified timing constraint will be imposed upon it. After this, the message will be delegated by the dispatch filter to the internal object *myNode*. Since this object is an instance of class *DistributionNode*, the message will pass through the filters of *DistributionNode* as well. In Figure 7, line (9), the real-time specification of *DistributionNode* is defined by invoking the method *endMin* on time, and therefore, the lowest timing value will be selected as a timing attribute of the message, which is $t_{consumerNew}$ in this case. The method *productionLineDisconnected* will automatically adjust its timing constraint with respect to $t_{consumerNew}$.

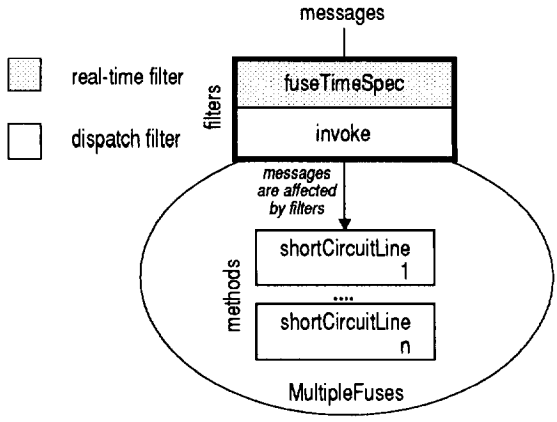



```
(1) class FastDistributionNode interface
(2)     begin
(3)         internals
(4)             myNode:DistributionNode;
(5)         filters
(6)             fastTimeSpec: RealTime = { True => self.* | time.minEnd (tconsumerNew | }
(7)             invoke: Dispatch = {True => myNode.* };
(8)     end;
```

Fig. 8. Interface definition of class *FastDistributionNode* which inherits from *DistributionNode* with a more restricted real-time constraint.

4.3. Eliminating the Non-Polymorphic Specifications Anomaly

Consider the interface definition of *MultipleFuses* as shown in Figure 9. Lines (3) to (6) declare the interface methods. The real-time filter *fuseTimeSpec* is declared in line (8). The element in this filter is defined by the specification "*.*", and therefore, all the received messages will match with this filter. The received message will take the value $t_{shortCircuit}$ as a timing attribute, if $t_{shortCircuit}$ is smaller than the timing value of the received message. The second filter *invoke* dispatches the received message for execution if this message matches with one of the methods defined by *MultipleFuses*.




```
(1) class MultipleFuses interface
(2)     begin
(3)         methods
(4)             shortCircuitLine1 returns nil;
(5)             shortCircuitLine2 returns nil;
(6)             ..... // list of shortCircuitLine methods for every remaining line //
(7)         filters
(8)             fuseTimeSpec: RealTime = { True => *.* | time.minEnd( t_shortCircuit) | };
(9)             invoke: Dispatch = {True => inner.* };
(10)    end;
```

Fig. 9. Interface definition of class *MultipleFuses*. The *RealTime* filter eliminates the *non-polymorphic specifications* anomaly.

The definition of class MultipleFuses does not suffer from the *non-polymorphic specifications* anomaly because the real-time specification as defined in line (8) is fully polymorphic and affects all the accepted messages[9] .

4.4. Eliminating the Orthogonally Restricting Specifications Anomaly

To eliminate the *orthogonally restricting specifications* anomaly, in Figure 10 we redefine the interface of *ProtectedDistributionNode*. This class declares two internals *myFuse* and *myNode* which are instances of class *MultipleFuses* and *DistributionNode*, respectively.

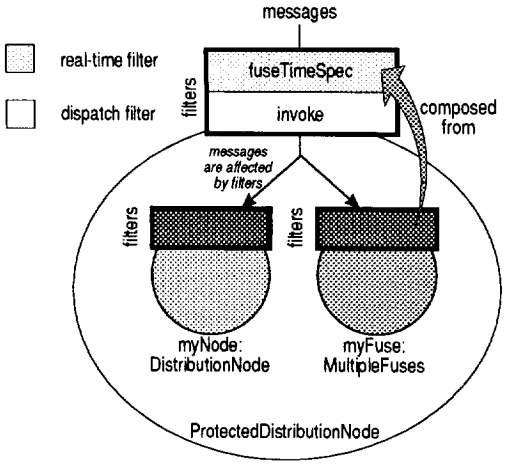
The first filter of *ProtectedDistributionNode* is inherited from *MultipleFuses*, by declaring it in line (6) as *myFuse.fuseTimeSpec*. Since *fuseTimeSpec* is declared at the interface of *myFuses*, it can be reused in class *ProtectedDistributionNode*. With every received message, this filter will associate the timing constraint of $t_{shortCircuit}$, if this value is smaller than the current timing value of the message.

The next filter is a dispatch filter and has the following semantics: the first element of the filter *myFuse.** specifies that all the incoming messages are delegated to the internal object *myFuse*, provided that these messages are supported by class *MultipleFuses*. If the first filter element does not match with the message, the second

---

9 Note that, instead of the wildcard "*.*", it is also possible to specify a set of messages on which the timing constraint should be applied.

filter element is evaluated. This filter element delegates the message to the internal object *myNode*, in case these messages are supported by class *DistributionNode*. The first filter element implements the inheritance from class *MultipleFuses* and the second filter element from *DistributionNode*. If a message is dispatched to one of these objects, then it will pass through the filters of the corresponding object as well. Since the message attribute is now $t_{shortCircuit}$, all the internal objects will take this value if $t_{shortCircuit}$ is less than $t_{consumer}$. As shown in Figure 7 lines, (10) and (11), the method *productionLineDisconnected* will adjust its value according to the lowest value.




```
(1) class ProtectedDistributionNode
interface
(2)     begin
(3)         internals
(4)             myFuse: MultipleFuses, myNode: DistributionNode;
(5)         filters
(6)             myFuse.fuseTimeSpec;
(7)             invoke: Dispatch = (True => myFuse.*, True => myNode.* );
(8)     end;
```

Fig. 10. The interface declaration of *ProtectedDistributionNode*. In this implementation the *orthogonally restricting* anomaly is eliminated.

4.5. Implementation Aspects

Currently, we are carrying out a research activity for the efficient implementation of real-time composition-filters. We are experimenting with a compiler that generates C++ and Smalltalk code. In most cases, real-time filters do not impose significant execution overhead. Consider for example, the definition of *ProtectedDistributionLine* as shown in Figure 10. Here, all the real-time filters are explicitly named at the interface of objects and the types of the internal objects are known at compile time. We intend to develop a translator to the FLEX language [13] because FLEX is supported by a measurement-based performance analyzer.

5. Conclusions

Object-oriented programming languages reduce the complexity of real-time applications through modularization so that *predictability* and *reliability* of applications can be increased. The inheritance mechanism can be useful in reusing well-defined and verified real-time programs. In most object-oriented real-time languages, however, there is a conflict between the inheritance mechanism and real-time specifications. This restrains the effective reuse of object-oriented programs with real-time specifications. We call this the *real-time inheritance* anomaly. In this paper we identified three kinds of anomalies: *mixing real-time specifications with application code*, *non-polymorphic real-time specifications* and *orthogonally restricting specifications*.

To overcome the real-time inheritance anomalies, we extend messages with a timing attribute. Real-time filters are used to affect this attribute. Since real-time filters separate real-time specifications from methods, programs written using real-time filters do not suffer from the *mixing real-time specifications with application code* and *non-polymorphic specifications* anomalies. The *orthogonally restricting* anomaly is eliminated through the compositional semantics of real-time filters. Although the real-time specifications were defined orthogonally, the semantics of filter composition are such that they are combined in an orthogonally restricting manner.

References

1. M. Aksit & A. Tripathi, *Data Abstraction Mechanisms in Sina/ST*, OOPSLA '88, pp. 265-275, September 1988.
2. M. Aksit, J.W. Dijkstra & A. Tripathi, *Atomic Delegation: Object-oriented Transactions*, IEEE Software, Vol. 8, No. 2, March 1991.
3. M. Aksit, L.Bergmans & S. Vural, *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, ECOOP '92, LNCS 615, pp 372-395, Springer-Verlag, 1992.
4. M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, *Abstracting Object-Interactions Using Composition-Filters*, in: *Object-based Distributed Processing*, R. Guerraoui, O. Nierstrasz & M. Riveill (eds.), to be published in the Lecture Notes in Computer Science series, Springer-Verlag, 1994
5. J. Allen, *Maintaining Knowledge about Temporal Intervals*, Communications of the ACM, Vol. 26(110), pp. 832-843, ACM, 1983.
6. L. Bergmans, M. Aksit, K. Wakita & A. Yonezawa, *An Object-Oriented Model for Extensible Synchronization and Concurrency Control,* Memoranda Informatica 92-87, University of Twente, January 1992.
7. E. Brorsson, C. Eriksson & J. Gustafsson, *RealTimeTalk: An Object-Oriented Language for Hard Real-Time Systems*, Proceedings of IFAC International Workshop on Real-Time Programming, 1992.
8. B. Dasarathy, *Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods for Validating Them*, IEEE Transactions on Software Engineering, 11(1), pp. 80-86, IEEE, 1985.
9. M. Ellis & B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

10. A. Goldberg & D. Robson, *Smalltalk-80 - The Language*, Addison-Wesley, 1989.
11. W. A. Halang & A. D. Stoyenko, *Constructing Predictable Real-Time Systems*, Kluwer Academic Publishers, Dordrecht-Hingham, 1991.
12. Y. Ishikawa, H. Tokuda & C.W. Clifford, *Object-Oriented Real-Time Language Design*, Carnegie Mellon University, USA, 1990.
13. K.J. Lin, J.W.S. Liu, K.B. Kenny & S. Natarajan, *FLEX: A Language for Programming Flexible Real-Time Systems*, Foundations of Real-Time Computing: Formal Specifications and Methods, pp. 251-290, (eds.) A.M. van Tilborg, G.M. Koob, Kluwer Academic Publishers, 1991.
14. 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
15. V.M. Nirkhe, S.K. Tripathi & A.K. Agrawal, *Language Support for the Maruti Real-Time System*, Proc. 1990 Real-Time Systems Symposium, pp. 257-266, IEEE Computer Society Press, 1990.
16. K. Schwan & H. Zhou, *Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads*, IEEE Transactions on Software Engineering, 18(8), pp. 736-748, 1992.
17. W.E.P. Sterren, *Design of a Real-Time Object-Oriented Language*, M.Sc. Thesis, Department of Computer Science, University of Twente, The Netherlands, Februari 1993.
18. K. Takashio & M. Tokoro, *DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems*, Proceedings of OOPSLA'92, pp. 276-294, ACM Press, 1992.
19. P. Wegner, *Dimensions of Object-Based Language Design*, pp. 168-182, OOPSLA '87, 1987.

Appendix A
Overview of Real-Time Specification Constructs in Object-Oriented Languages

The table below gives an overview of constructs for real-time specification as used in the selected object-oriented languages. Three categories of real-time specification constructs have been identified:

- *Periodic*: construct for specifying periodic executions.
- *Interval*: construct for specifying a time interval in which an execution needs to be terminated.
- *Time point*: constructs for specifying points in time at, before or after an event.

Language	Periodic	Interval	Time point
MPL	**every** *reltime* **do** **from** *abstime* **to** *abstime* **every** *time* **to**	**within** *reltime* **do** **from** *abstime* **to** *abstime* **do**	**at** *abstime* **start** **after** *abstime* **start** **before** *abstime* **start** **before** *abstime* **end** **after** *reltime* **start**
RTC++	**cycle** (*starttime*; *endtime*; *period*; *deadline*)	**within**(*time*)	**at** *time* **before** *time*
FLEX	none	B.**duration** <= *time* B.**duration** >= *time* B.**interval** <= *time* B.**interval** >= *time*	B.**start** <= *time* B.**start** >= *time* B.**finish** <= *time* B.**finish** >= *time*
RealTime-Talk	period time slot	maximal execution time slot	release time slot deadline slot
DROL	**active** *type methodname*(*params*) **start**(*time*) **end**(*time*) **period**(*time*) **deadline**(*time*) **timeout**(*time*)	*type methodname*(*param*) **within**(*time*) **timeout**(*exception*)	**invoke**(*target*)\{**time** *time1*:*method1***time** *time2*: *method2*\ ... \ **time** *timeN*:*methodN*}

Appendix B
Real-Time Specifications in Filters

In the current system, a message contains an object called *time*. This object consists of two attributes:

- *start*: the earliest start time of the message.
- *end*: the deadline of the message.

Times are specified relative to message acceptance, and are internally converted to an absolute time point.

To access the attributes, a set of methods are defined at the interface of object *time*:

- *minStart*(<*time point*>): the *start* attribute of the message is set to the earliest starting time, which is the minimum of the current value of the attribute and the argument *time point*.
- *maxStart*(<*time point*>): the *start* attribute of the message is set to the latest starting time, which is the maximum of its current value and the argument *time point*.
- *setStart*(<*time point*>): the *start* attribute is set to the argument *time point*.
- *getStart*: the *start* attribute is returned to the client.
- *minEnd*(<*time point*>): the *end* attribute of the message is set to the earliest deadline, which is either the current value of the attribute or the argument *time point*.
- *maxEnd*(<*time point*>): the *end* attribute of the message is set to the latest deadline, which is either its current value or the argument *time point*.
- *setEnd*(<*time point*>): the *end* attribute is set to the argument *time point*.
- *getEnd*: the *end* attribute is returned to the client.