Composing Synchronization and Real-Time Constraints

Lodewijk Bergmans and Mehmet Akşit
TRESE project, Department of Computer Science, University of Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands.
email: {bergmans | aksit}@cs.utwente.nl
ftp: ftp.cs.utwente.nl, directory: pub/doc/TRESE
www server: http://wwwtrese.cs.utwente.nl

Abstract

There have been a number of publications illustrating the successes of object-oriented techniques in creating highly reusable software systems. Several concurrent languages have been proposed for specifying reusable synchronization specifications. Recently, a number of real-time object-oriented languages have been introduced for building object-oriented programs with real-time behavior. Composing and reusing object-oriented programs with both synchronization and real-time constraints has not been addressed adequately, although most real-time systems are concurrent. This paper analyzes the origins of the problems in composing and reusing synchronization and real-time specifications, first as separate concerns, and later as composed behavior. To overcome the so-called inheritance anomaly problems, this paper proposes modular and composable synchronization and real-time specification extensions to the object-oriented model. The applicability of the proposed mechanisms is illustrated through a number of examples.

1. Introduction

Object-oriented analysis and design methods [50, 28] and programming languages like C++ [51] and Smalltalk [26] are now being applied to a large category of applications. Recently, a number of reusable object-oriented *design patterns* [24] have been introduced as a catalog of reusable object-oriented design knowledge. There have been a number of publications illustrating the successes of object-oriented techniques in creating highly reusable systems [45, 46]. These publications, however, largely deal with sequential systems without any consideration for concurrency and real-time aspects, although most *real-world* systems are concurrent and have some real-time aspect.

For several years, there have been many claims about the suitability of the object-oriented model for modeling concurrent systems. However, it appeared that extensibility and reusability of concurrent applications is far from trivial. The problems that arise, the so-called synchronization constraint inheritance anomaly, has been extensively discussed in the literature [35, 39, 15]. The inheritance anomaly means that introducing a new method and/or overriding an inherited method in a subclass may require additional redefinitions. Ideally, this should not be necessary. Several concurrent object-oriented languages [31, 22, 38] have been introduced to solve the synchronization constraint inheritance anomaly problem, although most of these languages suffer from this problem in one or other way.

Recently, there have been some attempts in defining real-time object-oriented languages [52, 17, 32, 27, 44]. These languages aim at reducing the complexity of applications through modularization so that *predictability* and *reliability* of applications can be increased. In addition, the inheritance mechanism can be useful in reusing well-defined and verified real-time programs. Similar to concurrent object-oriented languages, real-time object-oriented languages may suffer from the real-time constraints inheritance anomaly. In contrast to concurrent object-oriented languages, however, there has been almost no study on the origins of the real-time constraint inheritance anomaly problem. Needless to say, the combined analysis of concurrent and real-time constraint inheritance anomalies has not been addressed, although most real-time systems are concurrent.

It is important to note that both the synchronization and real-time constraint inheritance anomalies are not inherent with combining synchronization and real-time specifications, and inheritance. On the contrary, the anomalies are largely language dependent. The way how a language implements synchronization and real-time constraints, and inheritance can be the major cause of inheritance anomalies.

This paper has two contributions. Firstly, it presents generic object-oriented synchronization and real-time models which are useful in analyzing and relating synchronization and real-time constraint inheritance anomalies in a uniform way. These generic models help in searching for solutions that can

deal with both problems. Based on these generic models, a number of important synchronization and real-time inheritance anomalies are identified and discussed.

Secondly, as a possible solution to both synchronization and real-time constraint inheritance anomalies, this paper proposes modular and composable synchronization and real-time specification extensions to the object-oriented model using the concept of composition-filters. Composition-filters can affect the synchronization and real-time characteristics of the received and sent messages. By proper configuration of filters, one can specify synchronization and real-time constraints, and reuse of these constraints without causing inheritance anomalies. The applicability of the proposed mechanisms is illustrated through a number of examples.

The following section presents the example problem which is referred to throughout this paper. Section 3 and 4 define the issues in composing synchronization and real-time constraints, respectively. Section 5 makes a combined analysis of the synchronization and real-time constraint composition problems. In addition, this section presents a number of requirements for successfully composing synchronization and real-time specifications together. Section 6 introduces the application of the composition-filters concept in solving the problems presented in this paper. The related work is presented in section 7. Finally, section 8 evaluates our approach and gives conclusions.

2. An Example: A Software Controlled Crane System

We present a simple example to explain the issues of composing synchronization and real-time constraints in object-oriented programs. Our example is a simplified model of a software controlled crane that can lift and carry containers from arriving trucks to a buffer area. The containers are taken from this buffer area for further handling. To carry the containers, the crane uses a magnetic latching mechanism. Figure 1 depicts this system.

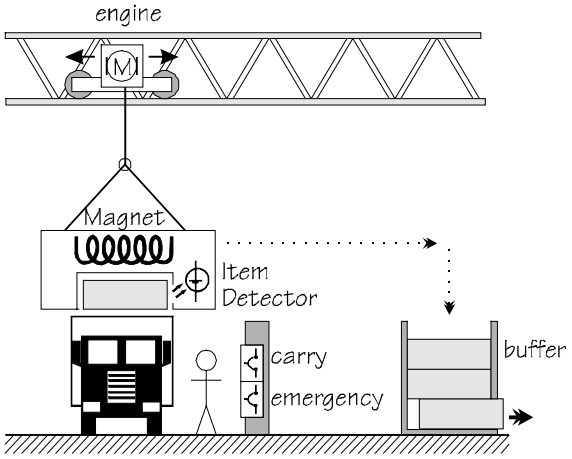

FIG. 1. A schematic overview of the crane system and its sensors.

The software is organized around an inheritance hierarchy which models different specializations of cranes. We will use an object-oriented pseudo-language for describing the examples; the meaning of the code examples should be self-explanatory.

Class *Crane* provides the basic functionality of the crane system, such as loading and carrying containers. In addition, class *Crane* stores and retrieves containers and inherits these buffering operations from class *BoundedBuffer*. In the following subsections, we will describe classes *BoundedBuffer* and *Crane*.

2.1 Definition of Class *BoundedBuffer*

Class *BoundedBuffer* implements a generic storage mechanism with limited capacity. It offers a simple protocol consisting of the message *get*, which retrieves the earliest stored element from the buffer, and the message *put(anElement)* which adds the argument *anElement* to the buffer. In our crane example, the message *put* is used to add a container to the buffer area, and the message *get* is to be invoked when retrieving a container from this area. Figures 2 and 3 show the definition of the interface of class *BoundedBuffer* and the effect of the *put* and *get* messages to the crane system. Here, the methods *Empty*, *Partial* and *Full* are defined for reading the state of the bounded buffer object.

The precise nature of the elements that are stored in a bounded buffer object is not relevant to this paper. In the definitions of methods *put* and *get* type *Any* is declared, which does not impose any type restrictions. The synchronization constraints of class *BoundedBuffer* will be discussed in section 3.

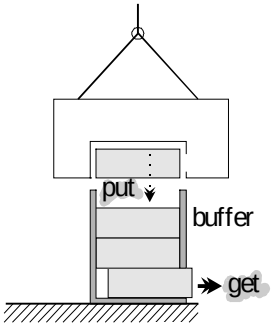

FIG. 2. The buffering of containers in the crane system.

```
class BoundedBuffer
methods
    size returns Integer;
    get returns Any;
    put(newElem:Any) returns Nil;
    Empty returns Boolean;
    Partial returns Boolean;
    Full returns Boolean;
end;
```

FIG. 3. Pseudo-code definition of class *BoundedBuffer*.

2.2 Definition of Class *Crane*

Typical operation of the crane consists of a fixed sequence of actions for picking up a container from a truck and putting it in the buffer area. We express this sequence by the following series of

messages to be sent to a crane object, for example as the result of pressing the *carry* button depicted in figure 1.

crane.on;	*latch the container by activating the magnetic field*
crane.forward;	*move the crane and container to the buffer area*
crane.put;	*offer the container to the buffer area*
crane.off;	*release the container by turning off the magnetic field*
crane.backward;	*move the crane backward to the truck*

Here, the method *put* is inherited from class *BoundedBuffer*. The definition of the interface of class *Crane* is shown in figure 4. Note that a number of *query methods* have been defined for reading the state of the crane object. For brevity, we have omitted method implementations.

```
class Crane
comment controls a crane installation;
inherits from BoundedBuffer;
methods
    on returns Nil comment turns on the magnetic field for latching items;
    off returns Nil comment turns off the magnetic field;
    forward returns Nil
        comment moves the crane forward until the buffer area has been reached;
    backward returns Nil
        comment moves the crane backward, until the loading spot has been reached;
    // the following are all query methods
    IsOn returns Boolean comment is the magnetic field on;
    IsOff returns Boolean comment is the magnetic field turned off;
    Moving returns Boolean comment is the crane moving;
    Stationary returns Boolean comment the crane is not moving;
    Loaded returns Boolean comment a container is loaded correctly;
    UnLoaded returns Boolean comment no container in place;
end;
```

FIG. 4. Pseudo-code definition of class *Crane*.

3. Issues in Composing Synchronization Constraints

This section studies the issues in composing synchronization constraints in concurrent object-oriented programs. First, the synchronization constraints of the crane system are specified. Second, a generic model is presented to analyze object-oriented synchronization composition mechanisms. Finally, possible synchronization inheritance anomalies are identified using the generic model.

3.1 Synchronization Constraints of the Crane System

To function properly, classes *BoundedBuffer* and *Crane* have to enforce certain synchronization constraints. Most object-oriented synchronization models define synchronization constraints using the states of objects and synchronization takes place only at the interfaces of objects prior to execution of methods. Referring to this model, we will describe in which state which messages should be blocked; in all other states the messages may proceed to execute.

3.1.1 Synchronization Constraints of Class *BoundedBuffer*

To avoid inconsistencies due to attempts of retrieving elements from an empty buffer or storing elements in a buffer that is already full, class *BoundedBuffer* must impose two synchronization constraints:

(S1) **if** Empty **then block** get;

This constraint specifies that if the buffer is empty (size=0), a received *get* message must be blocked until the buffer is no longer empty, which requires one or more invocations of *put* messages.

(S2) **if** Full **then block** put;

The constraint (S2) states that if the buffer is filled to its maximum capacity (size=limit), a received *put* message must be blocked until space becomes available. The latter requires the execution of one or more *get* messages.

3.1.2 Synchronization Constraints of Class *Crane*

This class introduces the following synchronization constraint (S3): The magnetic field may not be turned on as long as no container is in place to be latched.

(S3) **if** UnLoaded **then block** on;

The method *Unloaded* returns a *true* value if there is no container in place (as signaled by the *item detector* in figure 1). Note that, as soon as a container is put in place, a previously received and blocked *on* method will be activated.

The synchronization constraint of a class is the composition of the synchronization specifications of that class and all its superclasses. For class *Crane* this means that its synchronization constraint is the composition of (S1), (S2) and (S3).

3.1.3 Extending Class *Crane* with Class *ProtectedCrane*

For illustration purposes, we now introduce class *ProtectedCrane* which inherits from class *Crane* and introduces the synchronization constraint (S4):

(S4) **if** Moving **then block** on, off;

For safety purposes, the magnetic field is normally not allowed to be turned on or off while the crane is moving. The definition of the interface of class *ProtectedCrane* is shown in figure 5. Here, the synchronization constraint is specified in the **synchronization** clause.

```
class ProtectedCrane
    comment this class only introduces an additional synchronization constraint;
    inherits from Crane;
synchronization
    if Moving then block on, off;
        // do not turn the magnetic field on or off while the crane is moving;
end;
```

FIG. 5. Pseudo-code definition of class *ProtectedCrane*.

3.2 A Generic Model for Analyzing Synchronization Constraints

To identify the potential problems in inheriting concurrent code, in this subsection we make an attempt at defining a generic object-oriented synchronization model. Our starting point is the assumption of the conceptual object model depicted by figure 6. In this model, each object has an *object manager*, a *message queue*, a collection of methods and an object state. The object manager is responsible for scheduling: Based on the state of the object, the state of the message queue and the synchronization constraints, it decides whether a received message can result in immediate method execution, or whether it will be put in the message queue. The object manager also takes care that queued messages will eventually lead to the execution of a method, once their synchronization constraints are satisfied. The object manager operates on one message at a time to ensure that competing messages are handled consistently. One of the most significant properties of this model is that synchronization is performed after the messages arrive at the object, and just before the execution of the corresponding methods.

One can identify a large number of factors that -possibly- determine whether a message is acceptable or must be blocked. As stated before, message acceptance ultimately depends on the state of the object, in the broadest sense possible. We designate this by the term *implicit object state*, to avoid confusion with the common use of 'object state' which indicates the values of instance variables only. The implicit object state may include the values of instance variables, the properties of the message such as the message selector[1] and the message arguments, the current activities within the object (for example, the number of active threads), the message queue (i.e. what other messages are waiting to be served), and the history of the object (for instance, which messages have been executed previously and how often). A particular concurrent object-oriented language may express one or more of these, or may even offer other possible state information.

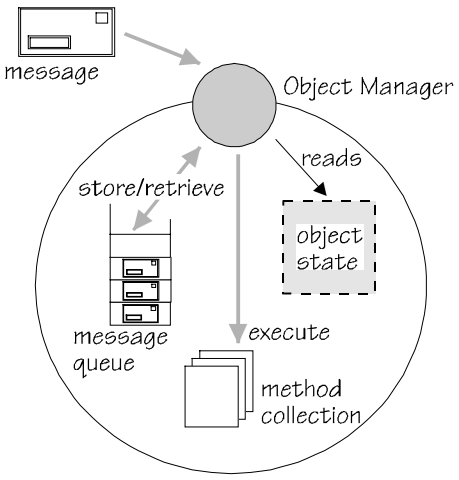

FIG. 6. A simplified model of the synchronization of messages.

[1] Message selector is a term used within the object-oriented community. It denotes the method to be invoked once the message is accepted.

The implicit object state is affected by several types of events: the arrival of new messages, the acceptance of a message or start of a message execution, the termination of a method execution, and the effects of method executions themselves. The implicit object state can be described as a region in a large state space. To make the implicit object state explicit in a program, *state abstraction* is required, resulting in a number of (synchronization) *conditions*. Each condition provides a concrete specification of the implicit state of the object.

For example, for a bounded buffer object, we can define the following three synchronization conditions:

Empty : self.size = 0

Partial : (self.size>0) and (self.size<limit)

Full : self.size = limit

Conditions have two important properties. Firstly, they perform extraction of the relevant state information, and secondly, they create an independence between the implementation-dependent implicit object state and the message-related synchronization code.

Synchronization of messages is modeled as a set of acceptable (non-blocked) messages, the so-called *accept set*. Synchronization can be specified by defining the mapping between the synchronization conditions and the set of possible accept sets of an object. The message accept set is offered to the scheduler, which uses it to activate –or postpone– the execution of messages. Figure 7 shows how the elements of the model are related.

A synchronization specification according to this model thus requires the following components:

1. *State Abstraction*: maps the implicit object state to synchronization conditions.
2. *Synchronization Condition Mapping*: maps the synchronization conditions to message acceptance.

Consider for example the language ACT++ [31]. In ACT++, synchronization is specified through so-called *behavioral abstraction*. For this purpose, every object defines a set of *behavior names* and for every behavior name the methods that are accepted for that behavior are listed. Every method specifies the next behavior with a *become* statement. The behavioral abstractions in ACT++ correspond to the synchronization conditions in our generic model. The *become* statements implement the state abstraction, and the definition of acceptable methods for each behavior realizes the condition mapping.

In some synchronization schemes, the mapping from the implicit object state to method acceptance is made immediately, without the intermediate step of synchronization conditions. This is the case in *guard-like* approaches, for example in the Guide language [20]. A method guard directly maps the current state of the object to message acceptance.

Most concurrent object-oriented languages that support interface control and object-level synchronization can be described by this model. We therefore use this generic model to describe the problems with reuse of synchronization code in a largely language independent way.

3.3 Synchronization Constraints Inheritance Anomalies

To avoid problems in reusing synchronization constraints through inheritance, the following requirements must be satisfied:

1. Modularity of synchronization specifications;
2. Composability of synchronization constraints;
3. Expressiveness for state abstractions.

The origins of inheritance anomalies can be traced back to ignoring one or more of these requirements, which are discussed in the following subsections.

3.3.1 Modularity of Synchronization Specifications

The modularity of synchronization specifications means that the synchronization specification is separated from the *application code* of the class. In fact, not only the synchronization-related specifications must be completely separated from the application code, but the synchronization specification must consist of an independent State Abstraction and Condition Mapping Functions, as depicted by figure 7. The prime reason for requiring synchronization modularity is to reduce the dependencies between the application code, such as method implementations, and the acceptance of messages. Because in subclasses the synchronization will virtually always change (if only to cope with newly added methods), dependencies between application code and message acceptance will require the redefinition of application code in many situations.

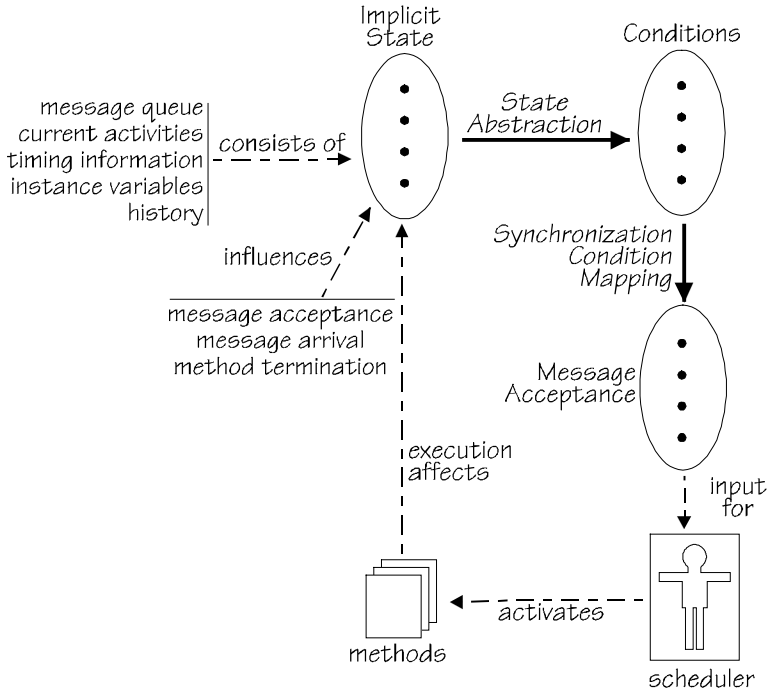

FIG. 7. Schematic outline of synchronization specifications.

3.3.2 Composability of Synchronization Constraints

The composability of synchronization constraints is an important factor for effective reuse and extension of synchronization constraints. The synchronization specification of a class must not be a *monolithic unit*. Specialization, extension or reuse of a monolithic synchronization specification in subclasses is not possible as it is not possible to replace parts of the specification. Adding new constraints to the specification is only feasible in specific cases (e.g. when the additional constraints are fully orthogonal to the existing specification).

If we consider the composability problem in our generic synchronization model, we must focus on Synchronization Conditions and on Synchronization Condition Mapping. Synchronization Conditions provide for an intermediate that is independent of both the implementation details of the object and the precise interface specification. They function as a reusable 'abstract synchronization specification' that does not violate encapsulation and can be tailored to the exact interface of the reusing client class[2]. This is because the synchronization conditions conform to logical states of the object with associated consistency constraints, such as *UnLoaded*, *Loaded*, *Stationary* and *Moving* for class *ProtectedCrane*.

For optimum composability, we require that every synchronization condition can be associated with the acceptance of multiple methods and that synchronization condition mappings can be synthesized into a single new condition mapping. We term the first requirement as *polymorphic synchronization specifications* and the second one as *synchronization synthesis*. These two requirements are discussed now.

Polymorphic synchronization specifications:

The requirement for polymorphic synchronization specifications is based on the goal of reusing and extending synchronization specifications. This requires that a certain (synchronization) constraint must be applicable to multiple methods, either in the same class or in subclasses.

Synchronization synthesis:

The term synthesis applies to the combination of multiple predefined components into a new one as well as the extension of a component with new, local, components. Synchronization synthesis is essential for the reuse and extension of synchronization constraints.

Consider, for example, the synchronization constraints of classes *Crane* and *ProtectedCrane*. Class Crane restricts the applicability of the method *on*; if there is no container in place, then the method *on* must be delayed. Class *ProtectedCrane* extends this restriction further. To execute the method *on*, the crane must be in a stationary state. These two constraints remain both valid in class

---

[2] Note that synchronization conditions do not imply a particular synchronization scheme; they can range from behavioral abstractions to guards.

ProtectedCrane, so what is needed is a *synthesis* of synchronization constraints. The problem here is that we need to chose the semantics for synthesizing the synchronization constraints. The most obvious is an AND semantics: only if both constraints are satisfied, the corresponding method will be activated. For certain problems, OR semantics may be required. This is discussed in section 8.1.4. Synthesizing synchronization constraints is even more difficult when combining two concurrent classes through multiple inheritance. We refer to [13] for more details on this topic.

In our generic synchronization model, the property of composable synchronization specifications is a characteristic of Synchronization Condition Mapping. In particular, with each synchronization condition multiple acceptable messages must be associated, and it must be possible to extend the mapping incrementally, in order to associate additional messages with an existing synchronization condition (as exemplified by the union operation on enabled sets in [53]). In figure 8, the condition mapping is shown for the constraints specified in classes *Crane* and *ProtectedCrane*.

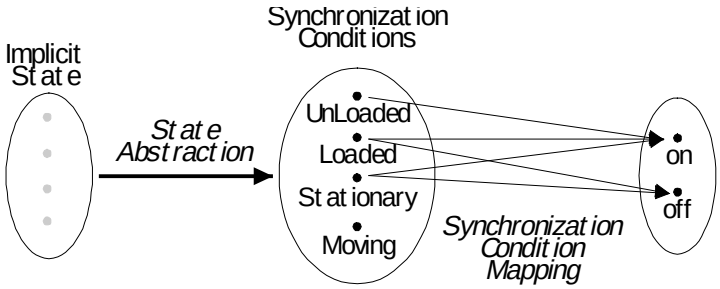

FIG. 8. Example of composability in the generic model.

3.3.3 Expressiveness for State Abstractions

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

One particular example of this is the so-called *history-sensitiveness* [35, 37], which is exemplified by an extension to the bounded buffer. Suppose that we want to add a subclass of *BoundedBuffer* that provides the method *gget*. This is an ordinary *get* operation, except that it should not execute immediately after a *put.* The problem with this example is that the state abstraction specification must be able to express the history (of received messages) of the object. This can be provided by the system (cf. the *wait_once* transitions in [38]), or the application must do some explicit bookkeeping.

In the *gget* example, assume that there is no special language support for keeping a history administration. Then the only way to know -inside a synchronization specification- that the latest executed method was a *put*, is by modifying all the methods of the class to maintain a Boolean status variable, say *justPut*.

Note that we can use the implementations of the superclass methods by using, for example, the pseudo-variable *super* as defined in the Smalltalk language. However, this does not make the

inheritance anomaly less severe. Not only must all methods be redefined (including the ones inherited indirectly), but in the subclasses the *justPut* Boolean must be maintained as well by all methods!

The relevance of this example is to demonstrate how lack of expressiveness can cause inheritance anomalies. This can be avoided through the ability to make a mapping from *any* implicit object state to a synchronization condition. The problem, however, is that the notion of implicit object state is quite broad and in fact system-dependent.

4. Issues in Composing Real-Time Constraints

In this section, first the real-time constraints of the crane system are defined. Second, a generic model is introduced to analyze object-oriented real-time specification techniques. Finally, with the help of this generic model real-time inheritance anomalies are identified.

4.1 Real-Time Constraints in the Crane System

We will now continue extending the inheritance hierarchy of the crane system by introducing new classes that specify real-time constraints. For brevity, we only show deadlines on method invocations and neglect other real-time specifications such as specifying starting times and periodicity[3].

4.1.1 Definition of Class *RTCrane*

Class *RTCrane* is an extension of class *ProtectedCrane* and has three modes: *Normal*, *Speed* and *HighSpeed*. If an instance of class *RTCrane* is in *Normal* mode, then no real-time constraints are enforced. If the instance is in *Speed* mode, then all its interface methods (both local and inherited) have to be completed in t_{speed} time units. If the instance is in *HighSpeed* mode, then all the interface methods have to be completed in $t_{highSpeed}$ time units, where $t_{highSpeed} < t_{speed}$. The instances of class *RTCrane* can be put into various modes to cope with the changing demands of its operating environment. Increasing the speed of a crane increases the energy consumption and restricts the maximum allowable weight of the containers to be carried. The modes of an instance can be changed dynamically. The definition of class *RTCrane* is shown in figure 9.

---

[3] Specifying starting times and periodicity does not introduce additional problems in real-time compositions. The reader can refer to [9] and [19] to obtain more information about real-time specification techniques.

```
class RTCrane
comment this class defines (two different) real-time constraints that apply to all methods;
inherits from  ProtectedCrane ;
methods
    setNormal returns Nil comment disable real-time constraints;
    setSpeed returns Nil comment set speed mode: enforce the deadline t_speed;
    setHighSpeed returns Nil comment set HighSpeed mode: enforce the deadline t_highSpeed;
    Normal returns Boolean comment in normal speed mode;
    Speed returns Boolean comment speed mode;
    HighSpeed returns Boolean comment in high speed mode;
realtime
    if Normal then before infinite end *;
    if Speed then before t_speed end *;
        // if the object is in speed mode, all methods have a deadline that is set to t_speed
    if HighSpeed then before t_highSpeed end *;
        // if the object is in HighSpeed mode, all methods have a deadline set to t_highSpeed
end;
```

FIG. 9. Pseudo-code definition of class *RTCrane*.

The mode of the object can be changed to *Normal*, *Speed* and *HighSpeed* modes, by invoking the methods *setNormal*, *setSpeed* and *setHighSpeed*, respectively. The real-time constraints are specified in the **realtime** clause. Depending on the condition specified between the **if** .. **then** clause, the construct **before** <*deadline*> **end** <*method(s)*> specifies that the execution of all <*method(s)*> is to be terminated before the time unit <*deadline*>. The deadline *infinite* indicates no deadlines is imposed. A wildcard '*' is used to designate all the interface methods of the object including the inherited methods.

4.1.2 Definition of Class *EmergencyRTCrane*

As a final example, consider class *EmergencyRTCrane* which inherits from class *RTCrane* and introduces the method *emergency*. The method *emergency* immediately shuts off the magnetic field, no matter what the current state of the crane is. This method can be invoked as the result of pressing an emergency button. The interface of class *EmergencyRTCrane* is defined in figure 10.

```
class EmergencyRTCrane
comment   this class introduces an emergency method: when it is invoked, it immediately drops the
          container and raises alarm signals. A deadline t_emergency is associated with the
          emergency method;
inherits from RTCrane ;
methods
    emergency returns Nil
        begin self.off; /* and generate alarm signals, set to urgent mode, .. */  end;
realtime
    before t_emergency end emergency;
end;
```

FIG. 10. Pseudo-code definition of class *EmergencyRTCrane*.

4.2 A Generic Model for Analyzing Real-Time Constraints

In this subsection we make an attempt at defining a generic object-oriented real-time model. We assume that timing information *travels together* with the message. In this way, timing information imposed by the caller of a message is made available to the receiver, which may additionally modify

the timing information. The application of timing constraints may depend on the actual state of the object (this includes the timing information of the current message). This is shown in figure 11.

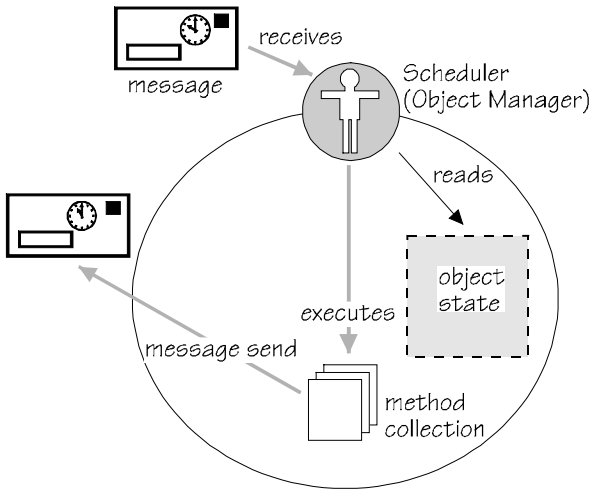

FIG. 11. A simplified model of the real-time scheduling of messages.

The generic model for real-time constraints shares parts with the generic model for synchronization discussed in the previous section: It is based on Implicit State, which is mapped through state abstraction to Real-Time Conditions. Real-Time Conditions are the necessary conditions for applying the Real-Time Constraints.

The combination of Real-Time Condition Mapping and Real-Time Constraints results in Message Timing Constraints, which define -optional- timing constraints for each message. The result of the message timing constraints and the available timing information of the message is offered to Scheduler[4], which uses these to schedule and activate method executions. The elements involved in the specification of real-time constraints are shown in figure 12.

A real-time constraint specification according to this model thus requires the following components:

1. State Abstraction: maps the implicit object state to real-time conditions.
2. Real-time Condition Mapping: maps the real-time conditions to message timing constraints.

Real-Time Condition mapping differs from Synchronization Condition Mapping through the fact that it includes timing specifications.

4.3 Real-Time Specification Inheritance Anomalies

The discussion of real-time constraint inheritance anomalies is very similar to the discussion of synchronization constraint inheritance anomalies, only the components of the model differ somewhat.

---

[4] The concept of real-time scheduling can be considered in a broader sense; scheduling corresponds to any possible implementation of real-time constraint specifications, including mapping to parallel machines and choosing among alternative implementations.

Similar to the requirements given in section 3.3, to avoid problems in reusing real-time constraints through inheritance, the following requirements must be satisfied:

1. Modularity of real-time specifications;
2. Composability of real-time constraints;
3. Expressiveness for state abstractions and timing specifications.

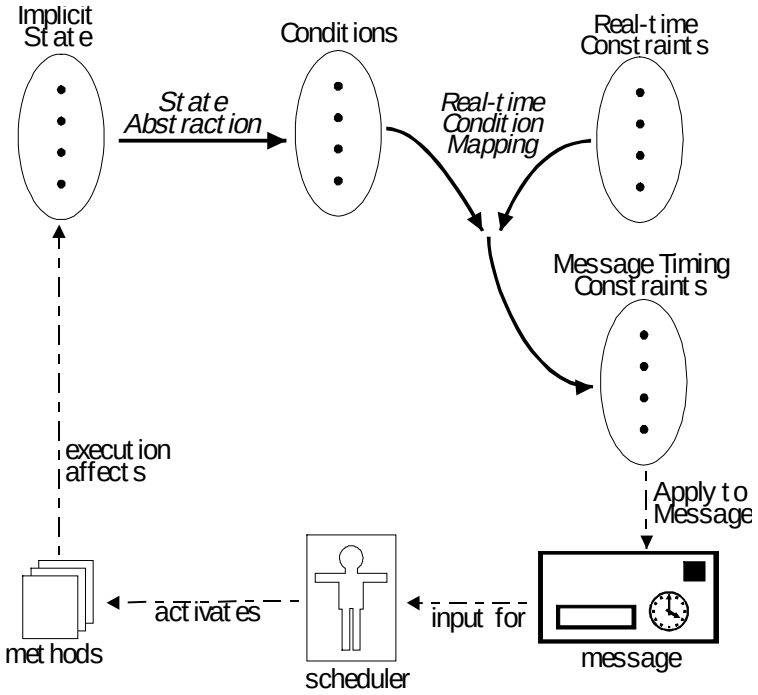

FIG. 12. A generic model for real-time constraints.

4.3.1 Modularity of Real-Time Specifications

The prime requirement to avoid inheritance anomalies is that the language provides modular specifications for State Abstraction and for Real-Time Constraint/Condition Mapping. If the real-time specifications cannot be separated from the method implementations, it is impossible to redefine the real-time specification nor the method implementations without redefining both. In addition, to avoid implementation dependencies, the implementation of the choice of (or between) real-time constraints must be separated as well.

4.3.2 Composability of Real-Time Constraints

We will study the real-time composability problem in two parts: the polymorphism requirement and the synthesis requirement.

Polymorphic real-time specifications:

In section 4.1.1, class *RTCrane* was defined with a number of real-time constraints that were applicable to all its methods. In general, real-time object-oriented languages do not provide such a specification construct, and therefore, the same real-time specification must be defined repeatedly. This causes two kinds of problems. Firstly, the intuitive real-time specification states that the deadline is imposed on *all* methods of the class. This should be also true when additional methods

are introduced in subclasses. This implies that all subclass implementors must be aware of this real-time constraint and add it to every newly defined method. Secondly, assume that a deadline is to be modified (either in the same class or in a subclass), then all the real-time constraint specifications must be updated.

Real-time synthesis:

The combination of real-time constraints from separate classes e.g. through (multiple) inheritance, may cause semantic interference, which will require additional redefinitions. As an example, assume that we modify class *EmergencyRTCrane* such that it imposes a certain deadline upon all methods of the class when the class is in the *Emergency* state. In this case, there is a potential conflict with the real-time constraints defined in class *RTCrane* for the *Speed* and *HighSpeed* modes. This may require -depending on the specific real-time constraint specification scheme- redefinition of the real-time constraints.

4.3.3 Expressiveness for State Abstractions and Timing Specifications

The reason that lack of expressiveness for state abstractions and for timing constraints causes inheritance anomaly is the same as in the discussion for synchronization inheritance. For example, assume that an object's real-time constraints depend on the history of the executions of some inherited methods. The problem here is similar to the one discussed in section 3.3.3; the state abstraction specification must be able to express the history (of received messages) of the object.

5. Issues in Composing Synchronization and Real-Time Constraints

5.1 Synchronization and Real-Time Compositions in the Crane System

In Table I, we illustrate the inheritance hierarchy of the crane system including both the synchronization and real-time constraints in the form of a table. In the left-most column the inheritance hierarchy is shown. Each class -except class *BoundedBuffer*- inherits from the class in the row above it. For the proper functioning of all classes, the inheritance mechanism of the adopted language must be able to compose the synchronization and real-time specifications properly; all constraints must be satisfied without redundant redefinitions of methods and constraints.

class hierarchy	synchronization constraint	real-time constraint
BoundedBuffer	*(S1)* empty: **block** get *(S2)* full: **block** put	–
Crane	*(S3)* unloaded: **block** on	–
ProtectedCrane	*(S4)* moving: **block** on, off	–
RTCrane	synchronization - real-time conflict	*(R1)* speed: **before** t_{speed} **end** * *(R2)* highSpeed: **before** $t_{highSpeed}$ **end** *
EmergencyRTCrane	synchronization - real-time conflict	*(R3)* **before** $t_{emergency}$ **end** emergency

Table I. The crane classes with their synchronization and real-time constraints.

Classes *BoundedBuffer* and *Crane* introduce synchronization constraints on different methods.

Classes *Crane* and *ProtectedCrane* both specify -complementary- constraints for the same method *on*. This indicates that the inheritance mechanism must be able to compose synchronization constraints defined for the same method.

ProtectedCrane specifies the same synchronization specification for the methods *on* and *off* (S4). An extreme example of such a polymorphic synchronization constraint would be: "full: **block** *". For example, if the buffer area is full, then all operations of the crane could have been blocked by this synchronization constraint.

The real-time constraints of class *RTCrane* (R1) and (R2) may conflict with the synchronization constraints of the methods *on*, *off*, *put* and *get*, which are defined by classes *BoundedBuffer*, *Crane* and *ProtectedCrane*. If one or more of these methods are blocked due to synchronization constraints, then the real-time constraints as specified by (R1) and (R2) may not be satisfied.

Class *EmergencyRTCrane* defines a real-time constraint for the method *emergency*; if the method *emergency* is invoked, then the magnetic field must be turned off immediately. However, class *ProtectedCrane* delays *off* messages while the crane is moving. The real-time constraint of *EmergencyRTCrane*, therefore conflicts with the synchronization constraint of *ProtectedCrane* if the method *emergency* is invoked while the crane is moving.

The synchronization constraints are defined to preserve the *logical behavior* of objects. For example, if the magnetic field is turned off while the crane is moving, then the container may fall down and be damaged.

A *real-time system* is a system in which the correctness of its behavior 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 even be harmful. The process is not allowed to execute outside the specified time-bound. For example, in the crane system if the method *emergency* is invoked because the crane operator is in danger, then carrying the container must be stopped immediately. *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.

5.2 Requirements for Composing Synchronization and Real-Time Constraints

From the discussion in the previous sections, we infer several requirements which object-oriented concurrent and real-time languages should fulfill, in order to be considered suitable for constructing extensible and reusable applications:

- *Eliminating inheritance anomalies through separation of concerns*: The adopted object-oriented language must enable the software engineer to define and reuse synchronization and real-time

constraints separately or together without causing inheritance anomalies. In section 3.2, Implicit State, Synchronization Conditions, Message Acceptance, Scheduler and Methods are defined as the essential elements of the generic synchronization model. The mapping functions State Abstraction, Synchronization Condition Mapping, and Method Executions relate these elements to each other. The generic real-time model extends the synchronization model with Real-Time Condition Mapping and Real-Time Constraints. Some of these elements and functions are implicit, such as Implicit State, Scheduler and Message Executions. The basic requirement for eliminating inheritance anomalies is that the adopted language must separate the synchronization and real-time elements and mapping functions from each other. In addition, the inheritance mechanism of the language must be able to compose these along the inheritance hierarchy.

- *Expression power*: The adopted language must provide sufficient expression power to deal with different synchronization and real-time requirements. For example, the State Abstraction Function must be expressive enough to represent various implicit states. Similarly, the Real-Time Condition Mapping and Real-Time Constraints must be able to specify different real-time requirements.

- *Assign priorities to specifications*: If synchronization and real-time specifications conflict with each other, then the software engineer must be able to assign priorities to synchronization and real-time constraints. While assigning priorities, the software engineer may consider two important cases:

  1. *Synchronization constraints are likely to prevail soft timing constraints*: Defining priorities can be based on analyzing critical states. Relatively the most unwanted state must be avoided. Consider the conflicts between real-time and synchronization constraints of classes *RTCrane* and *ProtectedCrane*. If the method *off* is invoked on an instance of class *RTCrane* and this instance is in *HighSpeed* mode and is moving, then the synchronization constraint of *ProtectedCrane* will defer the request until the crane is at *Stationary* state. Giving higher priority to real-time constraint may increase the speed but will possibly result in damage to the container. In such a case, it is preferable that the synchronization constraint has a higher priority than the real-time constraint. Note that the real-time constraints of *RTCrane* are soft timing constraints.

  2. *Hard timing constraints are likely to prevail synchronization constraints*: Consider, for example, the following situation. The method *emergency* is invoked because the crane operator is in danger and the further movement of the container would make the situation even worse. Turning off the magnetic field and possibly damaging the container is definitely preferable to risking the life of the operator. Note that the real-time constraints of *EmergencyRTCrane* are hard timing constraints.

6. The Composition Filters Approach

In this section, first we will present an object-oriented model to express composition of synchronization and real-time constraints. This model is based on the *composition-filters* concept. Second, we will model the crane system using this approach.

6.1 Extending the Object-Oriented Model with Composition Filters

In section 5.2, separation of concerns was presented as one of the fundamental requirements in eliminating inheritance anomalies. The conventional object-oriented model as adopted by languages like C++ and Smalltalk provides features that are very useful for a large category of applications and therefore it must be kept as a useful abstraction mechanism. However, to solve the identified problems[5], the object-oriented model must be enhanced modularly without losing its basic features. For this purpose, the composition-filters model extends the conventional model through a set of so-called *filters*[6].

Each message that arrives at an object is subject to evaluation and manipulation by the filters of that object. In figure 13, the manipulation of messages by the filters of an object is depicted. Each filter deals with a particular concern. In composing synchronization and real-time constraints there are basically three concerns:

1. Composition of synchronization constraints, handled by *Wait* filters;
2. Composition of real-time constraints, handled by *RealTime* filters;
3. Inheriting methods and executing messages, handled by *Dispatch* filters.

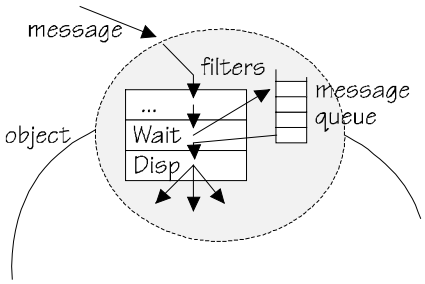

FIG. 13. The manipulation and buffering of messages at the filters of an object.

---

[5] As well as other identified problems such as the obstacles defined in [6].

[6] The composition-filters approach is a modeling paradigm rather than the definition of a particular language with fixed semantics. Composition-filters can be attached to objects expressed in different languages, such as Sina [2, 3], C++ [25] and Smalltalk [21]. The semantics of objects expressed in the composition-filters model can be largely determined by the semantics of the filters. Several different filter types have been defined in the past. For example, [2] illustrated how both inheritance and delegation can be simulated using filters. In [4], filters were introduced for defining reusable transactions. Language-database problems were addressed in [5]. In [7], filters were used to abstract coordinated behavior among objects. The application of composition-filters for real-time specifications was published in [8].

The aim of the composition-filters model is to improve the expression power of the conventional object-oriented model through modular and orthogonal extensions rather than building increasingly complex object structures. In the following sections, we will illustrate how composition-filters can be applied to compose synchronization and real-time constraints without creating inheritance anomalies.

6.2 Synchronization in the Composition-Filters Model

In figure 14, we show the interface definition of class *BoundedBuffer* expressed using the Sina language. The major extensions to the conventional object-oriented model are *condition* and *filter* declarations, as highlighted by gray boxes. Class *BoundedBuffer* declares the conditions *Empty*, *Partial* and *Full*, which correspond to the three states of a bounded buffer. The second extension is the declaration of two filters following the **inputfilters**[7] clause. A filter determines whether a particular message can be *accepted* or *rejected*. It also determines what action is to be performed in either case. Each filter is declared as an instance of a filter class. An arbitrary number of filters may be declared in a class definition. Class *BoundedBuffer* declares two filters, instances of class *Wait* and class *Dispatch*. This is because a bounded buffer object has two concerns: synchronization, handled by the wait filter and method execution, handled by the dispatch filter.

```
class BoundedBuffer interface
comment implements a bounded buffer with synchronization;
conditions
   Empty; Partial; Full;
methods
   put(Any) returns Nil;
   get returns Any;
inputfilters
   bufferSync : Wait = { Empty=>put, Partial=>{put, get}, Full=>get ;
   execute : Dispatch = { inner.* };
end      // class BoundedBuffer interface
```


FIG. 14. Sina definition of the interface of class *BoundedBuffer*.

A filter of class *Wait* has the following intuitive definition: When a message arrives at a wait filter, it can only proceed when the message is accepted by the filter, and it will be blocked otherwise, until the message *can* be accepted by the filter. In class *BoundedBuffer*, the input filter *bufferSync* of class *Wait* is declared as:

```
bufferSync : Wait = { Empty=>put, Partial=>{put, get}, Full=>get };
```

When a message is received by an instance of class *BoundedBuffer*, it is first applied to the wait filter. This wait filter has three parts separated by the operator ‘,’. The first part ‘Empty=>put’ has the following meaning: If the received message has the selector *put* and the condition *Empty* evaluates to *true*, then the message matches this part and will be forwarded to the dispatch filter.

---

[7] In addition to input filters, the composition-filters model also supports output filters. Output filters affect outgoing messages, whereas input filters affect incoming messages. For brevity, we do not further consider output filters in this paper.

Otherwise, the message is evaluated by the second part. The second part matches the message if the message has the selector *put* or *get*, and if the condition *Partial* evaluates to *true*. If both parts do not match, then the received message is evaluated by the third part expressed as 'Full=>get'. This part matches the message if the condition *Full* evaluates to *true* and if the message has the selector *get*. If the received message does not match any of these parts, then it will be blocked until one of the parts will match the message.

The *Dispatch* filter defines a simple matching mechanism that maps all received messages to the local ('*inner*') methods. The wild card '*' implies that all message selectors –that are supported by the object (inner)– are acceptable. The *Dispatch* filter takes care that a message that does match will be dispatched. Dispatching to a local method results in execution of that method.

To illustrate the synchronization of messages, we show the message queue for an instance of class *BoundedBuffer* that subsequently receives the messages *put*, *get*, *get'*, *get"* and *put'*. We assume the following condition implementations for class *BoundedBuffer*:

```
conditions
    Empty begin return noOfElements=0  end;    // no elements in the buffer
    Partial begin return (Empty.not and Full.not)  end;     // not empty, not full
    Full begin return noOfElements=maxSize;  end;       // the buffer is full
```

The subsequent states of the message queue of the object resulting from the arrival, buffering and dispatching of messages are shown in figure 15: As the first message *put* is received (no. 1), it is matched against the wait filter *bufferSync*. The message *put* matches the first filter element, as the *Empty* condition is valid. Thus the message is immediately dispatched and therefore removed from the queue again (no. 2). Now the buffer contains one element, causing condition *Partial* to become true. So when the message *get* arrives (no. 3), it is accepted by the filter, and immediately dispatched (no. 4). Condition *Partial* is no longer valid, causing the received message *get'* to be blocked (no. 5). When the next message *get"* arrives, it is placed in the queue, after the message *get'* (no. 6).

When *put'* arrives, it will be placed at the tail of the queue (no. 7), but since it is the first acceptable message in the queue, it will be dispatched prior to the *get* messages (no. 8). After *put'* has completed its execution (as we assume mutual exclusion here, which is enforced by default), condition *Partial* is valid once again. This enables both *get'* and *get"*, in which case the first applicable message (*get'*) in the queue is dispatched. After the execution of *get'*, the condition *Empty* will become *true* and condition *Partial* will be *false*, therefore *get"* remains in the queue (no. 9).

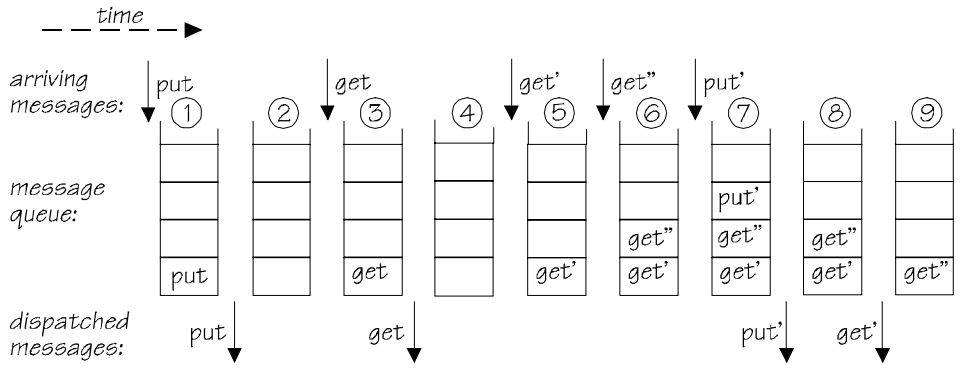

FIG. 15. Example of message scheduling in a bounded buffer.

6.2.1 Definition of Class *Crane* Using Filters

Now we will illustrate how class *Crane* can be implemented using the composition-filters approach. Consider the interface definition of class *Crane* in figure 16. The methods that allow reading the state of the object are declared following the **conditions** clause. Further, class *Crane* declares *container* of class *BoundedBuffer* as an *internal object*. Internal objects are used to implement composition-based inheritance as will be explained in the following paragraphs.

```
class Crane interface
internals
    container : BoundedBuffer;
conditions
    IsOn comment is the magnetic field on;
    IsOff comment is the magnetic field turned off;
    Moving comment is the crane moving;
    Stationary comment is the crane not moving;
    Loaded comment items are loaded;
    UnLoaded comment no items loaded;
methods
    on returns Nil comment turns on the magnetic field for latching items;
    off returns Nil comment turns off the magnetic field;
    forward returns Nil comment starts moving the crane forward;
    backward returns Nil comment starts moving the crane backward;
filters
    sync : Wait = { Loaded=>on, True~>on };
    compose : Dispatch = { inner.*, container.* }
end;
```

FIG. 16. Sina definition of the interface of class *Crane*.

Class *Crane* declares two filters: *sync* of class *Wait* and *compose* of class *Dispatch*. If a message is received by an instance of class *Crane*, then it will be first applied to the wait filter. This filter has two parts. The filter part 'Loaded=>on', means that the message with the selector *on* can match this part only if the condition *Loaded* evaluates to *true*. If the message cannot match this part, then it will be evaluated with respect to the next part. The second part 'True~>on' has the following meaning: All the received messages can match this expression except the message with the selector *on*. If the received message has the selector *on* and the condition *Loaded* is *false*, then it will be blocked until the condition *Loaded* becomes *true*. If the message matches the wait filter, then it will proceed to the dispatch filter. Otherwise the message will be blocked until it can be accepted by the wait filter.

The dispatch filter has two parts and does not define any conditions. The first part 'inner.*' has the following meaning. If the selector of the received message corresponds to one of the methods of class *Crane*, then the message will be dispatched to that method. If not, then the message will be evaluated with respect to the second part 'container.*'. Note that *container* is an internal object and an instance of class *BoundedBuffer*. If the selector of the received message corresponds to one of the methods of class *BoundedBuffer*, then it will be forwarded to *container*. Before dispatching to one of its methods, the message has to pass through the filters of *container* as well. This means that the received message has to fulfill the synchronization conditions of both classes *Crane* and *BoundedBuffer*.

This example illustrates how composition-filters can be used to compose synchronization specifications defined by different classes. Notice that the dispatch filter of class *Crane* implements an inheritance mechanism since the methods of class *BoundedBuffer* are now available at the interface of class *Crane*. This mechanism is also known as *composition-based inheritance*.

6.2.2 Definition of class *ProtectedCrane* Using Filters

Class *ProtectedCrane* inherits from class *Crane* and introduces a new synchronization constraint which restricts the execution of the methods *on* and *off*. Since the method *on* is also restricted by class *Crane*, the synchronization constraints of *Crane* and *ProtectedCrane* must be composed together for this method. The interface definition of class *ProtectedCrane* is shown in figure 17. To implement composition-based inheritance, class *ProtectedCrane* declares *myCrane* of class *Crane* as an internal object. Further, *ProtectedCrane* inherits all the conditions declared by *Crane* by using the expression 'myCrane.*' in the **conditions** clause.

```
class ProtectedCrane interface
comment this class only introduces an additional synchronization constraint:
    the magnetic field may not be turned on or off while the crane is moving;
internals
    myCrane : Crane; // an instance of the superclass Crane
conditions
    myCrane.*; // makes all the conditions of class Crane available for ProtectedCrane
filters
    protect : Wait = { Stationary=>{on, off} True~>{on, off} };
    inherit : Dispatch = { myCrane.* }; // inherit all methods from class Crane
end;
```

FIG. 17. Sina definition of class *ProtectedCrane*.

Class *ProtectedCrane* declares two filters. The first filter *protect* is an instance of *Wait* and is used to enforce the synchronization constraint of *ProtectedCrane*. If a message is received by an instance of class *ProtectedCrane*, it will be first evaluated with respect to the filter part 'Stationary=>{on, off}', which has the following meaning: If the received message has one of the selectors *on* or *off* and if the condition *Stationary* evaluates to *true* then this part matches the message. If the received message does not match, then it will be evaluated with respect to the second part 'True~>{on, off}'. This part accepts all the messages except the messages with the selectors *on* and *off*. Therefore, the

on and *off* messages can pass this wait filter only if the condition *Stationary* is true. Otherwise they will be blocked until the condition *Stationary* changes to *true*.

The dispatch filter *inherit* has only one part 'myCrane.*'. A message that arrives at the dispatch filter is forwarded to the instance *myCrane* of class *Crane* if the selector of the message corresponds to one of the interface methods of class *Crane*. Before dispatching to one of these methods, the message has to pass through the filters of classes *Crane* and *BoundedBuffer*, and therefore has to satisfy all the synchronization constraints of these classes as well[8].

6.3 Composing Real-Time Constraints Using Filters

6.3.1 Definition of Class RTCrane Using Filters

Class *RTCrane* inherits from class *ProtectedCrane* and depending on the *mode* of its instance object, it introduces two real-time constraints. The interface definition of class *RTCrane* is defined as depicted by figure 18. Class *RTCrane* declares an internal object *myCrane* of class *ProtectedCrane*. The three conditions *Normal*, *Speed* and *HighSpeed* are declared following the **conditions** clause.

```
class RTCrane interface
comment this class defines (two different) real-time constraints that apply to all methods;
internals
    myCrane : ProtectedCrane ;
conditions
    Normal comment in normal speed mode;
    Speed comment speed mode;
    HighSpeed comment in high speed mode;
methods
    setNormal returns Nil comment disable real-time constraints;
    setSpeed returns Nil comment set speed mode: enforce the deadline t_speed;
    setHighSpeed returns Nil comment set HighSpeed mode: enforce the deadline t_highSpeed;
filters
    rt:RealTime= {Speed=>* | time.setMin(t_speed)|, HighSpeed=>* |time.setMin(highSpeed)| };
    inherit : Dispatch = { inner.*, myCrane.* };
end;
```

FIG. 18. Sina definition of the interface of class *RTCrane*.

Class RTCrane declares two filters: *rt* and *inherit* which are instances of classes *RealTime* and *Dispatch*, respectively. The composition-filters model assumes that timing information travels with each message. The real-time filter is used to affect the timing attribute of message if the corresponding message matches the filter. If the message does not match the filter, then it will pass to the next filter without being affected by the filter.

---

[8] In certain cases, it is necessary to enforce synchronization constraints by wait filters atomically at a single class. To specify atomicity, a filter from a 'superclass' can be inserted, e.g. by declaring myCrane.sync' in the filter set of class *ProtectedCrane*. In our crane example, however, the atomic enforcement of synchronization constraints was not found necessary.

The real-time filter of class *RTCrane* has two parts. The first part is specified as 'Speed=> * | time.setMin(t_{speed}) |'. Here, if the condition *Speed* is true, then this part matches all the possible messages. If the message matches this part, then it will receive the timing constraint expressed between the separators '| ... |'. If the message does not match the first part, then it will be evaluated with respect to the second part of the real-time filter. The condition *Speed* is set to true by invoking the method *setSpeed*. In addition, this method assigns *false* to the other conditions *Normal* and *HighSpeed*.

The real-time constraint in the first part of the filter is expressed as | time.minEnd(t_{speed}) |. 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). If the condition *Speed* evaluates to true, then the real-time filter of class *RTCrane* associates with every message the timing value $t_{speed.}$ if t_{speed} is smaller that the current timing value of the received message.

The second part of the real-time filter is declared as 'HighSpeed=> * | time.setMin(highSpeed) |'. If the condition *HighSpeed* evaluates to true, then every message is associated with the timing value $t_{highSpeed}$, provided that this value is smaller that the current timing value of the received message. The condition *HighSpeed* is set to *true* by invoking the method *setHighSpeed*. The invocation of this method will make the other conditions *false*.

If the method *setNormal* is invoked, then the conditions *Speed* and *HighSpeed* are set to *false*. As a result, the received message cannot match the real-time filter and therefore will be forwarded to the dispatch filter without modifying the timing attribute.

The dispatch filter makes the methods of class *ProtectedCrane* available at the interface, provided that the received message can pass through the filters of class *ProtectedCrane*. If, for example, the condition *HighSpeed* is true, the received message has the selector *off* and the crane is moving, then this message will pass through the real-time and dispatch filters, but will be blocked by the wait filter of class *ProtectedCrane*. This means that in this particular implementation, the synchronization constraints have a higher priority than the real-time constraints. Since class *RTCrane* introduces soft timing constraints, we consider this priority assignment preferable.

6.3.2 Definition of Class *EmergencyRTCrane* Using Filters

Class *EmergencyRTCrane* inherits from *RTCrane* and introduces a hard timing constraint for the method *emergency*. If invoked, the method *emergency* must be executed within its time limit independent of other real-time and synchronization constraints. Consider the interface definition of class *EmergencyRTcrane* that is depicted in figure 19. To implement composition-based inheritance, class *EmergencyRTCrane* declares *myCrane* as an instance of class *RTCrane* following the **internals**

clause. Further, class *EmergencyRTCrane* declares the condition *Stationary* and 'inherits' all the conditions defined by *RTCrane* using the expression 'myCrane.*'. Note that the condition *Stationary* was declared before by class *ProtectedCrane*. Redefinition of a condition overrides the previous definition. The purpose of this redefinition will be explained in the following paragraphs.

```
class EmergencyRTCrane interface
comment this class introduces another real-time constraint with a higher priority;
internals
    myCrane : RTCrane ;
conditions
    Stationary  // this overrides the Stationary condition defined in class Crane;
    myCrane.*; // make all (other) conditions provided by RTCrane available;
methods
    emergency returns Nil;
filters
    emergency : RealTime = { emergency | time.setMin(t_emergency) | };
    inherit : Dispatch = { inner.*, myCrane.* };
end;
```

FIG. 19. Sina definition of the interface of class *EmergencyRTCrane*.

Class *EmergencyRTCrane* defines two filters: a real-time filter and a dispatch filter. The real-time filter has only one part. If the received message has the selector *emergency*, then it will receive the real-time deadline $t_{emergency}$ provided that $t_{emergency}$ is smaller than the current deadline. All other requests will be passed to the dispatch filter without being affected by this real-time filter. The dispatch filter either dispatches the message to the method *emergency* or to the internal object *myCrane* of class *RTCrane*. The implementation of class *EmergencyRTCrane* is shown in figure 20.

```
class EmergencyRTCrane implementation
conditions
    Stationary
        comment this condition overrides the Stationary condition such that in case the sender of
            the message is equal to the server (i.e. the first receiver), the condition always returns
            true, otherwise the existing condition implementation is used;
        begin return (client=server) or myCrane.Stationary end ;
methods
    emergency
        begin
            server.off;
            // generate warning signal, e.g. alarm bell
        end;
end;
```

FIG. 20. Sina definition of the implementation of class *EmergencyRTCrane*.

The method *emergency* carries out two kinds of actions. First, it turns off the magnetic field by invoking the method *off* on *server*. In the Sina language, *server* denotes the receiver of the message and it is comparable to the pseudo-variable *self* in Smalltalk or *this* in C++. Further, the Sina language provides the pseudo variable *client* which denotes the sender of a message. After shutting off the magnetic field, the method *emergency* generates warning signals.

Now assume that the received message has the selector *emergency* and has no timing constraints. The instance of class *EmergencyRTCrane* will then assign the timing attribute $t_{emergency}$ to the

message and forward it to the dispatch filter. The dispatch filter will execute the method *emergency*. The method emergency will then invoke the method *off* on server, which is the same instance that received the emergency request. This self invocation will be forwarded by the dispatch filter to the internal object *myCrane* of class *RTCrane*. Assume that the crane has been put in *HighSpeed* mode previously. Then the real-time filter of *RTCrane* will try to assign the timing attribute $t_{highSpeed}$ to the message. However, since $t_{emergency}$ is smaller than $t_{highSpeed}$, $t_{emergency}$ will be kept as the timing attribute of the message. The internal object *myCrane* of class *RTCrane* will further forward the message to its own internal object which is an instance of class *ProtectedCrane*. If the crane is moving, then the wait filter of class *ProtectedCrane* will normally block the *off* and *on* requests, because its condition *Stationary* will be *false*. This would give a higher priority to the synchronization constraints, which is not desired for class EmergencyRTCrane.

To give a higher priority to the real-time constraint of the method *emergency* than the synchronization constraints of class *ProtectedCrane*, the condition *Stationary* is redefined by class *EmergencyRTCrane*. The condition *Stationary* evaluates to *true* if the message is a self call (client=server), or the condition *Stationary* evaluates to *true* by the internal object *myCrane* (myCrane.Stationary). Polymorphically redefining conditions allows programmers to give different priorities to constraints along the inheritance hierarchy.

7. Related Work

In the following, we will present the related work first in concurrent, and later in real-time object-oriented languages.

7.1 Concurrent Object-Oriented Languages

In this section, we will give a list of significant examples of concurrent object-oriented languages. In the following subsections, we will evaluate these languages using the three requirements presented in section 3.3.

We observe four distinct approaches in object-oriented synchronization schemes. These are namely, *explicit acceptance*, *activation conditions*, *meta-level* and *inter-object level* synchronizations. Explicit acceptance means that during a method execution, the state of the object is considered, and based on this it is explicitly decided what message to accept next. For example ABCL/1 [55], POOL [10] and Eiffel// [18] support this approach.

Activation conditions define for each message whether it should be accepted or blocked. In some languages the activation conditions can be manipulated directly (e.g. Actor languages [1] and Hybrid [42]). In other languages and systems the activation conditions are specified indirectly, e.g. through guards (e.g. in [22], Guide [20] and Concurrency Annotations [33]), path expressions (e.g. PROCOL [14]) or state abstractions (e.g. ACT++ [31], Rosette [53] and Synchronizing Actions [41]). The proposal in [38] combines activation conditions through guards with state abstractions.

Concurrent reflective languages define synchronization constraints at a meta-level. Examples of this are ABCL/R [54], Actalk [16] and MAUD [23].

Another approach towards specifying synchronization is at the inter-object level; synchronization problems can be solved by controlling the patterns of interactions between objects. Although such approaches generally do not suffer from inheritance anomaly, they inflict with modularity and the notion of autonomous objects, thereby causing other problems in reusability and extensibility. Examples of inter-object level synchronization specifications are atomic transactions [40], atomic delegations [4], Abstract Communication Types [7] and Synchronization Patterns [34].

7.1.1 Definition of the Synchronization Inheritance Anomaly Problem

During the last several years, various researchers [10, 15, 30, 43] have indicated that there is an interference between inheritance and concurrency. The identification and description of inheritance anomalies was first made by Matsuoka, Yonezawa and Wakita in [35] and [37]. Basically, this work classified the inheritance anomalies with respect to representation and manipulation of object states. Here, the anomalies were termed as 'mixing synchronization specification with application code', 'state-partitioning', 'orthogonally restricting specifications', 'history- sensitiveness', etc.

Our definition of anomalies is based on the generic synchronization model presented in section 3.2. Our analysis is somewhat more general. For example, in section 3.3.1, we generalized the 'mixing synchronization specification with application code' anomaly to the problem of 'non-modular synchronization specifications'. We claimed that the modularity definition must not only include the separation of synchronization specification from application code, but also that the synchronization specification must consist of an independent State Abstraction and Condition Mapping Functions, as depicted by figure 7. In fact our generic model made it possible to compare the synchronization-related inheritance problems to the problems of inheriting real-time constraints.

7.1.2 Evaluation of Languages with Respect to Synchronization Inheritance Anomalies

Modularity of synchronization specifications

In a number of concurrent object-oriented languages with explicit message acceptance, each object defines a part called the *body*; this is a piece of code that is executed independently from the execution of incoming messages (e.g. POOL-I [11], Eiffel// [18]). In these systems, the body represents the high-level specification of a process that is associated with the object: the received messages are considered and when deemed appropriate, the execution of corresponding methods is initiated. It is obvious that the body approach requires complete redefinition of the body in a subclass whenever the synchronization changes. In particular, in the subclass all the synchronization constraints for the methods defined in parent classes must be defined again. This conflicts with incremental specification. It also violates encapsulation, because synchronization constraints that are

implementation-dependent must be redefined as well. This creates a dependency between the subclass and the superclasses: redefining the synchronization of the superclass requires (all) the subclasses to update the body implementations.

In addition to realizing the synchronization constraints of the object, the body may implement some general house keeping operations as well. Modifying these operations (i.e. application code) cannot be done independently from synchronization, thus requiring additional unwanted redefinitions.

Composability of synchronization constraints

The requirement for polymorphic synchronization specifications states that a certain synchronization constraint must be applicable to multiple methods, either in the same class or in subclasses. The languages that adopt guard-like approaches such as Guide [20], Concurrency Annotations [33] and Synchronisers [38][9] cannot effectively specify polymorphic synchronization specifications.

In some languages the objective of synchronization synthesis has been discarded, usually motivated by the complexity of the problem and the relatively small amount of reused code. Examples of this are POOL-I [11], Eiffel// [18] and Dragoon [12, 48]. In these languages synchronization specifications can be inherited (once), but not extended afterwards.

Expressiveness for state abstractions

The expressive power for specifying state abstractions varies widely among synchronization schemes. At one end of the scale are mechanisms such as path expressions, which can only express acceptance based on very specific information (i.e. the history of executed messages). Extended path expressions provide guards to extend the expressive power. Another specific mechanism for expressing synchronization is based on synchronization counters [49]. The more sophisticated synchronization schemes (e.g. in [22] and [38]) allow -almost- the full expressiveness of the programming language for expressing synchronization conditions.

A solution to the bookkeeping problem of maintaining history information as defined in section 3.3.3 is available in some systems through the definition of generic pre- and post-actions that are executed for all messages that are received by the object (e.g. in Encapsulators [47], or in reflective languages such as ABCL/R [36] and MAUD [23]).

It should be noted that the approaches based on behavioural abstraction that perform a *become* statement at, or after the end of methods (e.g. ACT++ [31]) are obliged to predict the state of the object at the moment of message arrival. Thus they cannot deal with any information about arrived messages in the time between the *become* and the (test for) acceptance of a particular message.

---

[9] Although this is less severe because their approach supports behavioral abstractions as well, which can cope with this particular problem.

7.2 Real-Time Object-Oriented Programming Languages

In the following paragraphs, we will present some significant real-time object-oriented languages. The evaluation of these languages will be presented in the next subsection.

The Maruti programming language (MPL) [44] is based on the C++ language 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[10] to the server object and to the methods called by it. MPL offers several synchronization constructs that are orthogonal to the object structure and are embedded in the method bodies of an object.

RTC++ [27] 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 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. RTC++ offers a guard-like synchronization mechanism, declared at the interface of every method.

The Flex language [32] 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. Synchronization must be specified with the constraints embedded in method implementations as well.

RealTimeTalk [17] is based on the Smalltalk language 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. RealTimeTalk cannot specify explicit synchronization constraints.

The DROL language [52] 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

---

[10] 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.

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.* The server chooses from a set of alternatives 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. Synchronization in DROL is based on the enabled-set approach [53].

7.2.1 Definition of the Real-Time Inheritance Problem

To the best of our knowledge, apart from our related work [8], real-time constraints inheritance anomalies have not been studied in the literature. Our definition in [8] is somewhat different from the analysis that is presented in this paper. In [8], real-time constraint inheritance anomalies were classified as 'mixing real-time specifications with application code', 'non-polymorphic real-time specifications' and 'orthogonally restricting real-time specifications'. The 'mixing real-time specifications with application code' anomaly is a special case of the modularity anomaly described in section 4.2.1. The 'non-polymorphic' and 'orthogonally restricting' real-time specification anomalies correspond to respectively the polymorphic real-time specifications and real-time synthesis anomalies described in section 4.3.2. The analysis technique presented in this paper is based on the generic real-time model and is more general. In addition, the generic model made it possible to analyze synchronization and real-time inheritance anomalies together.

7.2.2 Evaluation of Languages with Respect to Real-Time Inheritance Anomalies

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

The Maruti Programming language is only capable of expressing timing and synchronization constraints within methods; timing and synchronization constraints are always encapsulated and cannot be separately inherited nor reused in subclasses. Therefore, MPL suffers from both the modularity and the composability inheritance anomalies in both the synchronization and the real-time domain.

In RTC++ timing constraints can be declared either at the method interface or in the method body. If the latter is the case, this will cause the modularity anomaly. Otherwise, the timing (and synchronization) constraints are defined per method header. As a result, the composability anomaly

applies as well. Expressing guards for synchronization in RTC++ has several limitations, resulting in the expressiveness anomaly.

Flex associates both synchronization and timing constraints with statements, or blocks of statements. Therefore the modularity requirement is not met, resulting in anomalies. Although blocks can be labeled for access by other constraints, the constraints themselves cannot be reused by other constraints, resulting in composability anomalies.

Because RealTimeTalk does not allow for subclassing of classes with real-time specifications, and also does not support explicit synchronization constraints, the notion of inheritance anomaly does not apply to RealTimeTalk.

DROL offers two distinct timing specification approaches. Timing constraints for message sends are embedded in the method body of the sender, causing modularity and composability inheritance anomalies. The (worst case) timing for message reception is declared at the interface of the object, and cannot be reused separately, causing composability anomalies. Time polymorphic invocations are powerful, but do not contribute directly to the composability of real-time constraints. The synchronization approach suffers from modularity anomaly, since the enabled sets are determined inside the method body.

8. Evaluation

In section 5.2, the requirements for effectively composing and reusing synchronization and real-time constraint specifications were summarized by three points: separation of concerns, expression power and assigning priorities to specifications. We will now evaluate the composition-filters approach with respect to these requirements.

8.1 Separation of Concerns

Separation of concerns is one of the essential requirements to compose and reuse synchronization and real-time constraint specifications. Within this context, the following four requirements were identified:

1. Separation of synchronization and real-time specifications and method executions;
2. Separation of the elements of synchronization specifications;
3. Separation of the elements of real-time specifications;
4. Composition of concerns.

8.1.1 Synchronization specification, Real-Time Specification and Method Execution

The composition-filters approach separates the concerns of specifying synchronization and real-time constraints and inheriting and executing methods by three different filters which are functionally independent of each other. Each filter affects the received messages in a certain way. For example, wait filters may delay messages, real-time filters may affect the timing characteristic of messages and

dispatch filters may execute the corresponding methods. Specification of filters are separated from the implementation of objects.

8.1.2 Elements of Synchronization Specifications

In section 3.2, the elements of our generic synchronization model were defined as Implicit State, Synchronization Conditions, Message Acceptance, Scheduler and Methods. The mapping functions State Abstraction, Synchronization Condition Mapping, and Method Executions relate these elements to each other. We will now analyze how elements of synchronization specifications are modeled within the composition-filters approach.

Consider, for example the specification of the wait filter of class Crane: {Loaded=>on, True~>on}. Here, the condition *Loaded* corresponds to the element Synchronization Condition. The element State Abstraction Function is defined by the implementation of the condition method *Loaded*. The operator '=>' corresponds to Synchronization Condition Mapping Function. The '~>' operator specifies an *exclusive* Condition Mapping Function. An exclusive Condition Mapping Function provides an open-ended specification of synchronization conditions. For example, the wait filter of class *Crane* only restricts the messages with the selector *on*. Message Acceptance is specified as a set. In the wait filter specification of class *Crane*, 'on' refers only to the method *on* of class *Crane*. A filter specification may refer to the total message interface of a class. For example, in the dispatch filter of Crane, the specification 'container.*' refers to the total message interface of class *BoundedBuffer*. The character '*' provides an open-ended specification of object's behavior which may be extended by new compositions.

Synchronization Condition, Synchronization Condition Mapping and Message Acceptance can be combined in various ways. For example, the specification 'Loaded=>on' is a one-to-one combination. The specification 'True~>on' is a one-to-many combination.

8.1.3 Elements of Real-Time Specifications

In real-time filters, most elements of real-time specifications are similar to synchronization specifications. The evaluation presented in the previous section is also valid for real-time filters. In addition, real-time filters express real-time constraints delimited by the character "|". Real-time constraints can be associated with a set of methods and conditions. For example, the specification 'Speed => * |time.setMin(t_{speed})|' relates the real-time constraint t_{speed} to all the messages received by the object and to the condition *Speed*.

8.1.4 Composition of Concerns

The composition-filters model allows compositions of separately specified constraints. For example, Class *ProtectedCrane*, as defined in section 6.2.1, composes the synchronization constraint of class *Crane* and class *ProtectedCrane* for the method *on*. This is the so-called AND composition of constraints. A received message has to pass through two different filters, thus satisfying the

conditions of both filters. An example for OR composition of constraints is illustrated by classes *BoundedBuffer* and *Crane*. These classes enforce synchronization constraints on different sets of methods. The OR composition in class *Crane* is defined through the usage of multiple parts in a filter. The operator ',', in fact, is a Conditional-OR composition operator. Using different filters provides separation of concerns of different application-domain requirements, such as concurrent and real-time systems.

8.2 Expression Power

As we outlined before in the discussion about inheritance anomalies, sufficient expression power for specifying the mapping functions is relevant for two reasons: firstly to be able to express certain aspects, such as deciding message acceptance based on the number of active threads in the object. Secondly, in some cases such a requirement can be expressed through extra code in the implementation of the object. As we have shown previously, such an approach will generally lead to inheritance anomalies. Implicit Object State covers a virtually unlimited state-space. Therefore, the most obvious area where expressive power may be lacking is in the State Abstraction Function. As the composition filters approach offers the full functionality of method implementations for expressing state abstraction mapping, most results can be defined in a straightforward manner. For example in the guards approach of Guide [20], the guard expressions are restricted to a logical expression over instance variables and synchronization counters. In addition, condition methods can be inherited or polymorphically redefined. For example, classes *ProtectedCrane* and *EmergencyRTCrane* inherit the condition methods of their superclasses. Class *EmergencyRTCrane* polymorphically redefines the condition of class *ProtectedCrane*. We should note that the redefinition of conditions may have unwanted consequences, as improper usage may violate the synchronization constraints from the superclass. On the other hand, the equivalent mechanism of polymorphic method overriding is widely accepted in object-oriented programming.

The expression power of real-time filters is equivalent to the expression power of most real-time languages. A detailed description of real-time specifications in the composition-filters model is given in [8].

8.3 Assign Priorities to Specifications

In the composition-filters model, a message may have to pass through the filters of several objects, before being dispatched to the corresponding method. Consider for example, class *RTCrane* which receives a message with the selector *on*. This message has to pass through the filters of RTCrane, ProtectedCrane and Crane before being dispatched to the method on. Real-time filters may affect the timing attribute of messages, whereas wait filters may block them. This means that in case of AND composition, the synchronization specification will prevail the real-time specification. This was illustrated in the definition of class RTCrane. As in the definition of class *EmergencyRTCrane*, to

give a higher priority to real-time specifications, the synchronization conditions must be polymorphically redefined.

8.4 Conclusion

Recently, a considerable number of concurrent and real-time object-oriented languages have been introduced. However, composing and reusing object-oriented programs with both synchronization and real-time constraints has not been addressed, even though most real-world systems are concurrent, and have some real-time aspects. In short, failing in providing composable synchronization and real-time constraints results in so-called synchronization and real-time inheritance, anomalies. This implies that the reuse and composition of real-time and concurrent objects is unnecessarily limited.

To analyze these inheritance anomalies, we presented generic object-oriented models for synchronization and real-time constraint specifications. These models are more general than the previously proposed ones, and led us to identify a set of requirements that are necessary to avoid the inheritance anomaly problem.

To satisfy the identified requirements we extended the object-oriented model modularly with composition filters. These modular and orthogonal extensions allow us to overcome both synchronization and real-time constraint inheritance anomalies. A number of examples using the Sina language were shown to illustrate the applicability of the proposed mechanism. For a more precise description of composition-filters and the Sina language we refer to [13, 29].

References

[1] G. Agha. An Overview of Actor Languages, *ACM SIGPLAN Notices*, Vol. 21,No. 10, October 86, pp. 58-67.

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

[3] M. Aksit. *On the Design of the Object-Oriented Language Sina*, Ph.D Thesis, University of Twente, 1989.

[4] M. Aksit, J.W. Dijkstra and A. Tripathi. Atomic Delegation: Object-oriented Transactions, *IEEE Software*, Vol. 8, No. 2, March 1991, pp 84-92.

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

[6] M. Aksit and L. Bergmans. Obstacles in Object-Oriented Software Development, *Proc. of the OOPSLA'92 Conference*, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 341-358.

[7] 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 791, Springer-Verlag, 1993, pp 152-184.

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

[9] J. Allen. Maintaining Knowledge About Temporal Intervals, Communications of the ACM, Vol.26, No. 110, ACM, 1983, pp. 832-843.

[10] P. America. POOL-T: A Parallel Object-Oriented Language, *In Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro (eds), The MIT Press, Cambridge, Mass. 1987, pp. 199-220.

[11] P. America. A Parallel Object-Oriented Language with Inheritance and Subtyping, *Proc. of the OOPSLA/ECOOP '90 Conference*, ACM SIGPLAN Notices, Vol. 25, No. 10, October 1990, pp. 161-168.

[12] C. Atkinson, S. Goldsack, A. Di Maio and R. Bayan. Object Oriented Concurrency and Distribution in DRAGOON, J. of Object-Oriented Programming, March/April 1991, pp. 11-18.

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

[14] J. van den Bosch and C. Laffra. PROCOL-- A Parallel Object Language with Protocols, *Proc. of the OOPSLA '89 Conference*, ACM SIGPLAN Notices, Vol. 24, No.10, 1989, pp 95-102.

[15] J-P. Briot and A. Yonezawa. Inheritance and Synchronization in Concurrent OOP, *Proc of the ECOOP'87 Conference*, Springer-Verlag, 1987, pp. 32-40.

[16] J.-P. Briot. Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment, *Proc. of the ECOOP '89 Conference*, Cambridge University Press, Nottingham, July 10-14, 1989, pp. 109-129.

[17] E. Brorsson, C. Eriksson and J. Gustafsson. RealTimeTalk: An Object-Oriented Language for Hard Real-Time Systems, *Proc. of IFAC International Workshop on Real-Time Systems*, Brugge, 1991.

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

[19] B. Dasarathy. Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods for Validating Them, IEEE Transactions on Software Engineering, Vol. 11, No. 1, 1985, pp. 80-86.

[20] D. Decouchant, P. Le Dot, M. Riveill, C. Roisin and X. Rousset de Pina. A Synchronization Mechanism for an Object-Oriented Distributed System, *Proc. of the 11th IEEE Conference on Distributed Computing*, May 1991.

[21] W. van Dijk and J. Mordhorst. *CFIST, Composition Filters in Smalltalk*, Graduation Report, HIO Enschede, The Netherlands, May 1995.

[22] S. Frølund. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages, *Proc. of the ECOOP '92 Conference*, LNCS 615, Springer-Verlag, 1992, pp. 185-196.

[23] S. Frølund and G. Agha. A Language Framework for Multi-Object Coordination, *Proceedings of the ECOOP '93 Conference*, LNCS 707, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 346-360.

[24] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[25] M. Glandrup. *Extending C++ Using the Concepts of Composition Filters*, M.Sc. Thesis, University of Twente, November 1995.

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

[27] Y. Ishikawa, H. Tokuda and C.W. Clifford. *Object-Oriented Real-Time Language Design*, Carnegie Mellon University, USA, 1990.

[28] I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard. *Object-Oriented Software Engineering -- A Use Case Driven Approach*, Addison-Wesley/ACM Press, 1992.

[29] P. Koopmans. *On the Definition and Implementation of the Sina/st Language*, M.Sc. Thesis, University of Twente, The Netherlands, July 1995.

[30] D.G. Kafura and K.H. Lee. Inheritance in Actor Based Concurrent Object-Oriented Languages, *Proceedings ECOOP '89*, Cambridge University Press, Nottingham, July 10-14, 1989, pp. 131-145.

[31] D.G. Kafura and K.H. Lee. ACT++: Building a Concurrent C++ with Actors, J. of Object-Oriented Programming May/June 1990, Vol. 3, No. 1, pp. 25-37.

[32] K.J. Lin, J.W.S. Liu, K.B. Kenny and 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.

[33] K.-P. Löhr. Concurrency Annotations, *Proc. of the OOPSLA '92 Conference*, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, 1992, pp. 327-340.

[34] C.V. Lopes and K. Lieberherr. Abstracting Process-to-Function relations in Concurrent Object-Oriented Applications, *Proc of the ECOOP '94 Conference*, LNCS 821, Springer Verlag, July 1994, pp. 81-99.

[35] S. Matsuoka, K. Wakita and A. Yonezawa. *Synchronization Constraints with Inheritance: What is Not Possible-So What is?*, Tokyo University, Internal Report, 1990.

[36] S. Matsuoka, T. Watanabe and A. Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming, *Proc. of the ECOOP '91 Conference*, LNCS 512, Springer-Verlag, 1991, pp. 213-250.

[37] S. Matsuoka and A. Yonezawa. Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, in *Research Directions in Concurrent Object-Oriented Programming*, (eds.) G. Agha, P. Wegner and A. Yonezawa, MIT Press, April 1993, pp. 107-150.

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

[39] C. McHale. *Synchronisation in Concurrent Object-Oriented Languages: Expressive Power, Genericity and Inheritance*, Ph.D. Thesis, University of Dublin, Trinity College, 1994.

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

[41] C. Neusius. Adapting Synchronization Counters to the Requirements of Inheritance, ACM SIGPLAN OOPS Messenger, Vol. 2, No. 4, October 1991.

[42] O.M. Nierstrasz. Active Objects in Hybrid, *Proc. of the OOPSLA '87 Conference*, ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 243-253.

[43] O. Nierstrasz. The Next 700 Concurrent Object-Oriented Languages -- Reflections on the Future of Object-Based Concurrency, in *Object Composition*, Centre Universitaire d'Informatique, University of Geneva, June 1991, pp. 165-187.

[44] V.M. Nirkhe, S.K. Tripathi and A.K. Agrawal. Language Support for the Maruti Real-Time System, *Proc. 1990 Real-Time Systems Symposium*, IEEE Computer Society Press, 1990, pp. 257-266.

[45] *Proc. of the OOPSLA'94 Conference*, ACM SIGPLAN Notices, Vol. 29, No. 10, October 1994.

[46] *Proc. of the OOPSLA'95 Conference*, ACM SIGPLAN Notices, Vol. 30, No. 10, October 1995.

[47] G.A. Pascoe. Encapsulators: A New Software Paradigm in Smalltalk-80, *Proc. of the OOPSLA '86 Conference*, ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986, pp. 341-346.

[48] S.C. Reghizzi, G.G. de Paratesi and S. Genolini. Definition of reusable concurrent software components, *Proc. of the ECOOP '91 Conference*, LNCS 512, Springer-Verlag, 1991, pp. 148-166.

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

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

[51] B. Stroustrup. *The C++ Programming Language*, Addison-Wesley, 1986.

[52] K. Takashio and M. Tokoro. DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems, *Proc of the* OOPLSA *'92 Conference*, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 276-294.

[53] C. Tomlinson and V. Singh. Inheritance and Synchronization with Enabled-sets, *Proc. of the OOPSLA '89 Conference*, ACM SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 103-112.

[54] T. Watanabe and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language, *Proc. of the OOPSLA '88 Conference*, ACM SIGPLAN Notices, Vol. 23, No. 11, November 1988, pp. 306-315.

[55] A. Yonezawa, E. Shibayama, T. Takada and Y. Honda. Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, in *Object-Oriented Concurrent Programming*, (eds.) A. Yonezawa, M. Tokoro, The MIT Press, 1987, pp. 55-89.