Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages

Svend Frølund

Department of Computer Science
1304 W. Springfield Avenue
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: frolund@cs.uiuc.edu

Abstract. We analyse how inheritance of synchronization constraints should be supported. The conclusion of our analysis is that inheritance of synchronization constraints should take the form of incrementally *more* restrictive constraints for derived subclasses. Our conclusion is based on the view that combinations of behavior in object-oriented languages yield subclasses that *extend* superclass behavior. We give a notation for describing synchronization constraints. In our notation, synchronization constraints can be inherited and aggregated. We present a number of examples that illustrate the fundamental concepts captured by our notation. Synchronization constraints are described as restrictions that apply to invocation of methods. Application of restrictions is *pattern-based*, which allows the same restriction to apply to multiple methods and multiple restrictions to apply to the same method.

1 Introduction

Synchronization constraints provide for data consistency on a per object basis in a concurrent system. Synchronization constraints specify the circumstances under which an object's methods may be invoked. A method that may be invoked according to the synchronization constraints is said to be **enabled**.

Inheritance is a key structuring mechanism in object-oriented languages. In concurrent object-oriented languages, it is desirable to inherit synchronization constraints whenever possible, to avoid reimplementation of superclass synchronization constraints in subclasses. The synchronization constraints associated with a given method (name) in a superclass may need to be changed in subclasses. In that case, inheritance of synchronization constraints requires that synchronization constraints can be **incrementally modified**, i.e. changed without being (re)implemented from scratch in subclasses.

Derivation of methods is one reason why synchronization constraints may need to be incrementally modified in subclasses. A derived method is one whose declaration results in a vertical name collision [8] and is a common phenomena in object-oriented programs. Virtual methods in Beta [9], virtual functions in C++ [18] and methods in Smalltalk [5] are all examples of derivable methods. A derived method has an ancestor method: the method in a superclass whose name is shadowed in a subclass. Derivation of methods means that a name bound to a given behavior (method

body) in a superclass may be bound to a different behavior in subclasses. Therefore, the synchronization constraints associated with that name may need to change in subclasses.

Integration of inheritance and synchronization constraints has been investigated in a number of recent papers [13] [4] [16] [12]. However, all of these proposals are insufficient since they do not support incremental modification of synchronization constraints. In all of the proposals, synchronization constraints are specified as activation conditions: a boolean expression associated with each method. The expression is evaluated prior to method invocations and if the expression evaluates to true, the invocation is legal. In the above proposals, activation conditions can not be incrementally modified or reused in the definition of other activation conditions. In particular, the activation condition of a derived method is always specified from scratch. In this paper we present a framework in which synchronization constraints can be incrementally modified and thereby inherited.

In Section 2, we present insights about the specification of synchronization constraints in object-oriented concurrent languages. We analyse how to express synchronization constraints so that they can be inherited. The analysis is conducted in a language independent manner. In particular, no specific inheritance mechanism is assumed. The framework presented in Section 3 builds on our analysis. In order to give our ideas a more concrete form, we provide a notation for describing synchronization constraints. The notation is deliberately kept simple to illustrate the concepts that we find important when specifying synchronization constraints. We give a number of examples that present the fundamental principles of our notation. Section 4 concludes and relates our framework to other approaches.

2 Analysis

Incremental modification of synchronization constraints should take a form that integrates well with the way in which object-oriented languages support combination of behaviors. Most object-oriented languages support derivation of methods that *extend* the behavior of ancestor methods. Extension of behavior is directly supported by the INNER construct used in Beta [10]. The `super` pseudo variable of Smalltalk gives a more flexible way of combining behaviors, but the semantics of `super` still supports extension of ancestor methods. For a more in-depth discussion of different ways of combining method behaviors, refer to [2].

Execution of a method that extends the behavior of an ancestor method might result in data inconsistency in situations where the ancestor would leave the object in a consistent state. Extension of behavior means that additional actions are executed and each additional action may potentially result in data inconsistency. Since the purpose of synchronization constraints is to prevent inconsistencies, a language that supports extension of behavior should also support incremental restriction of synchronization constraints. Therefore, incremental modifications should give *more restrictive* synchronization constraints. The synchronization constraints specified in a superclass should denote an *upper limit* on permissible method invocations in all subclasses.

An important aspect of inheritance is **factorization** of common properties into superclasses. Factorization means that properties described in superclasses will also

hold for subclasses. Factorization makes it possible to reason about class hierarchies in terms of the generic properties that hold for the (abstract) superclasses of the hierarchy.

Synchronization constraints denote properties that we refer to as **synchronization properties**. Synchronization constraints for a superclass should be specified in a way so that the resulting synchronization properties hold in all subclasses. Suppose that synchronization constraints are specified as conditions that *enable* methods. Then synchronization properties could be defined in terms of the following pseudo notation:

property `P` : **condition** `C` **enables** `Method`

However, property `P` will *not* hold in subclasses where the synchronization constraints for `Method` are made more stringent. Thus, if synchronization constraints are specified as conditions that enable methods, the resulting synchronization properties do not necessarily hold in subclasses and, as such, can not be factored out.

For synchronization properties to hold in subclasses, synchronization constraints must be specified as conditions that **disable** methods. Such synchronization constraints give rise to the following pseudo notation for synchronization properties:

property `P` : **condition** `C` **disables** `Method`

Properties of this kind hold for subclasses when the synchronization constraints for `Method` are made increasingly more restrictive. Methods that are disabled in a superclass will *always* be disabled in a subclass.

Although necessary, supporting incremental modification of synchronization constraints is not sufficient to guarantee their inheritance. If the behavior of a derived method totally redefines the behavior of its ancestor, it would be meaningless to inherit the synchronization constraints of the ancestor method. If redefinition of behavior is possible, redefinition of synchronization constraints should be possible.

In summary, we have argued that synchronization constraints should be specified in terms of disabling restrictions instead of enabling conditions. Furthermore, inheritance of synchronization constraints should yield more restrictive constraints in subclasses. In the following section we present a framework that captures these two principles.

3 A Framework for Synchronization Constraints

In this section we present the concepts and principles that we believe constitute an appropriate platform for describing synchronization constraints. The concepts and principles are made concrete in the form of a notation for describing synchronization constraints. Our notation only captures synchronization constraints. Other entities of a language such as classes and methods are not described since their concrete representation is immaterial to our framework.

Section 3.1 gives some assumptions on which the notation is based. In Section 3.2, the basics of our notation are introduced. Inheritance of synchronization constraints within our notation is illustrated in Section 3.3. Finally, Section 3.4 shows how synchronization constraints may be defined as aggregates of other synchronization constraints.

3.1 Assumptions for the Framework

It is immaterial to the framework whether invocations are synchronous or asynchronous. Invocation requests can arrive at an object any time and independently of the object's state. As depicted in Figure 1, part of each object is a **controller** that intercepts invocation requests arriving for that object. A controller is an active component of an object that makes sure that incoming invocation requests are scheduled according to the synchronization constraints of the object. If an invocation request can not be scheduled right away, it is put into a pending queue by the controller. The controller may attempt to (re)schedule invocation requests in the pending queue at a later point in time (e.g. after each method invocation).

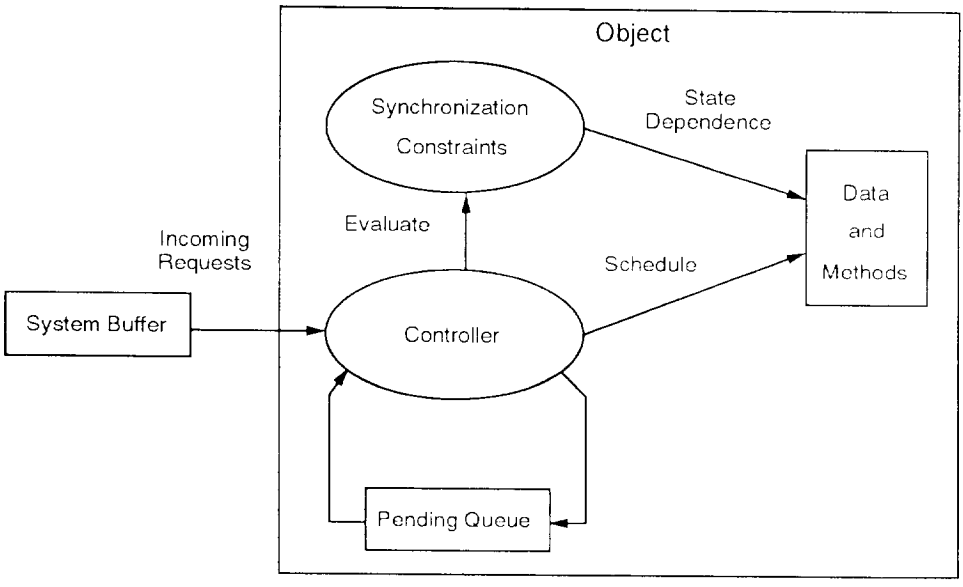

Fig. 1. The structure of active objects

To keep our notation as simple as possible, we ignore the following aspects of synchronization constraints:

- *Priorities and fairness.* We do not describe strategies for choosing between multiple invocation request that may all be scheduled according to the synchronization constraints.
- *Thread control.* We assume precautions are taken by the controller to control the number of concurrent threads running within an object. Thread control as part of the controller means that all objects employ the same kind of thread control and is the strategy used in Beta, PROCOL [22] and POOL-I [1]. In some languages, thread control is user-defined and often described within the constructs that define synchronization constraints for classes of objects. User-defined thread control is possible in Guide [4], Mediators [6] and Synchronizing

Actions [13]. We have ignored thread control for reasons of simplicity. However, in the conclusion we argue that user-defined thread control can in fact be smoothly incorporated into our framework.

Objects are instantiated from classes. Classes may be hierarchically organized according to an inheritance relation. Synchronization constraints are specified on per class basis and all objects instantiated from the same class have synchronization constraints with similar functionality. We assume that synchronization constraints are specified as a separate part of class descriptions. In fact, synchronization constraints is the only part of a class that is specified in our notation.

Having separate specification of synchronization constraints is generally desirable to avoid the "inheritance anomaly" [11] [12]. The anomaly manifests itself as subclasses in which specification of correct synchronization constraints require redefinition of method behavior that would otherwise be reusable. The anomaly may, for example, occur if synchronization constraints are specified as *part* of method behaviors. In that case, superclass synchronization constraints can not be changed in subclasses without also changing the methods of which the synchronization constraints are part. For a more elaborate discussion of the inheritance anomaly and its causes, refer to [12].

3.2 Notation

Our notation for describing synchronization constraints is given in Figure 2. Synchronization constraints define restrictions on the acceptance of invocation requests. Invocation requests are called "invocations" and they contain the name of the method to be invoked and the actual parameter values of the invocation.

$$
\begin{array}{ll}
i \in & \text{Invocations} \\
n \in & \text{Method names} \\
e \in & \text{Expressions} \\
v \in & \text{Values} \\
x \in & \text{Variables} \\
p \in & \text{Invocation patterns} \\
r \in & \text{Restrictions}
\end{array}
$$

$$
\begin{array}{l}
i ::= n(v_1, \ldots, v_m) \\
p ::= n(x_1, \ldots, x_k) \mid \textbf{all-except } p \mid n \\
r ::= e \textbf{ prevents } p
\end{array}
$$

Fig. 2. Abstract syntax for our notation

We assume arbitrary expressions and we only use simple arithmetic and boolean expressions in our examples. Furthermore, we assume that construction of invocations is an eager operation. There is nothing in our framework, however, that prevents expressions from being evaluated in a lazy fashion. As an example, the invocation

`Oper(4,5)` denotes an invocation of the method `Oper` with argument expressions that evaluate to the values 4 and 5.

A restriction applies to all invocations that match its pattern. Pattern-based application of restrictions means that the same restriction may apply to multiple methods and that multiple restrictions may apply to the same method. Pattern matching is defined according to these rules:

- The pattern $n(x_1, \ldots, x_k)$ matches invocations of a method with name n. Furthermore, the names $x_1, \ldots, x_k$ are bound to the values of a matching invocation.
- The pattern **all-except** p matches all invocations that do not match the pattern p.
- The pattern n is used in situations where a restriction does not depend on any of the actual parameter values.

The above way of describing patterns is deliberately made simple to focus on the synchronization aspects. More advanced pattern matching facilities may be desirable in a "real" language, such as the type-based pattern matching found in Linda [3]. Another extension would be to make patterns first class values so that patterns may be named and composed.

The expression associated with a restriction is called the **condition** of the restriction. A condition is a boolean expression evaluated in a context with the variables in the pattern bound to the values in a matching invocation. The evaluation context of conditions also includes the instance variables of the class in which the restriction is defined. Thus, the legality of a given invocation may depend on the actual parameter values of the invocation and the state of the object to which the request is made. Restrictions that depend on an object's invocation history can be expressed by introducing special instance variables that record the invocation history. A given invocation may cause multiple conditions to be evaluated, but we should not assume anything about the order in which conditions are evaluated. Thus, implementations of our framework should ensure that conditions are side-effect free.

A restriction is like a "negative" guard. When the condition of a restriction is true for an invocation that match the pattern of the restriction, the invocation may not be serviced. For an invocation to be legal, it may not match the pattern of any restriction with a true condition. It only takes one true condition to prevent an invocation from being serviced. Put formally:

$$\textbf{disabled}\ i = \mathit{condition}_1\ \textbf{or}\ \ldots\ \textbf{or}\ \mathit{condition}_n$$

Where $\mathit{condition}_1 \ldots \mathit{condition}_n$ are the boolean values that result from evaluating the condition-expressions of the restrictions whose pattern i matches. **disabled** is a predicate that gives the legality of accepting a given invocation.

Example 1. Consider the classical example of a bounded buffer. Using our notation, the synchronization constraints of a bounded buffer can be specified using these two restrictions:

```
( size ≥ MAX )  prevents put
( size ≤ 0 )    prevents get
```

`size` is an instance variable denoting the current number of elements in the buffer. `MAX` is a constant expression giving the maximum number of elements that a buffer can hold. The data consistency that must be maintained for the buffer is: $0 \leq$ `size` $\leq$ `MAX`. □

3.3 Inheritance of Restrictions

Synchronization constraints are specified in a way so that they can be incrementally modified. The restrictions applicable to a given method name in a superclass can be supplemented by additional restrictions for the same method name in subclasses. Thus, our framework supports the view that synchronization constraints become increasingly more stringent in subclasses. Example 2 and Example 3 below present situations in which synchronization constraints may be inherited.

Example 2. A simple example in which synchronization constraints can be reused is the following. Suppose a subclass with a `get-2` method is derived from the bounded buffer of Example 1. The semantics of `get-2` is to atomically *get* two elements from the buffer. This means that `get-2` is not enabled when there is 1 or 0 elements in the buffer.

(`size` $\leq$ `1`) **prevents** `get-2`

Note that the behavior of the `put` and `get` methods need not be changed in order to describe the synchronization constraints for the `get-2` method. □

Example 3. Consider the concept of a resource administrator. A number of client processes compete for a number of resources. In order to maintain properties like security, exclusive access, etc. the resources are accessed through an administrator object. In Figure 3, an inheritance hierarchy of resource administrators is depicted. In the following, we illustrate how synchronization constraints may be inherited in this hierarchy.

The class `resource-administrator` is most general. It defines two methods `access` and `free`. A client issues requests for resources through the `access` method. After use, a client puts a resource back to the pool of available resources by calling the `free` method. The consistency notion of the `resource-administrator` class is that it can not hand out more than the maximum number of resources. The synchronization constraints for `resource-administrator` are given by the restriction:

(`resources-given-out` $\geq$ `maximum-resources`) **prevents** `access`

The `different-resources` subclass administers and makes a distinction between different kinds of resources. For simplicity, this example just involves two kinds of resources: heavyweight and lightweight. An instance of the `different-resources` class is illustrated in Figure 3. Since heavy-weight resources are more expensive in terms of physical resources like memory, at most two such resources can be accessed. In order to distinguish between the different kinds of resource requests, the `access` and `free` methods are refined by derivation in the `different-resources` subclass. The synchronization constraints of the `resource-administrator` class also applies to the `different-resources` class: no more than the available resources can be

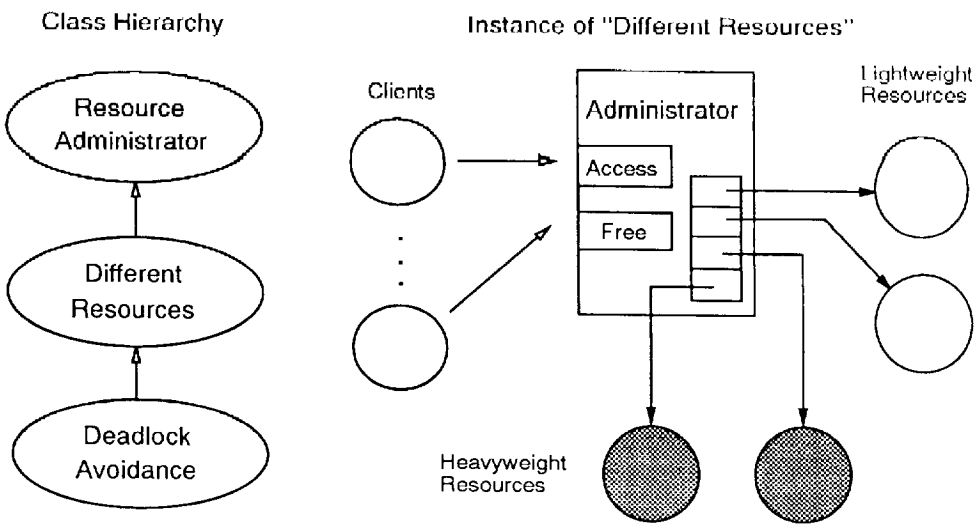

Fig. 3. Resource Administrators

accessed. In addition, the synchronization constraints of the `different-resources` should reflect the fact that only 2 heavyweight resources may be accessed. Thus, the `different-resources` subclass further restricts the invocations of the `access` method by this restriction:

```
( kind-of(request) = heavyweight and heavyweight-given-out = 2 )
prevents access(request)
```

The `deadlock-avoidance` subclass of `different-resources` runs some deadlock avoidance algorithm before granting a request for resources. Any deadlock avoidance algorithm suffice, but, to make the description more concrete, the bankers algorithm [17] is used. The methods of `different-resources` need not be changed in order to derive this subclass. The synchronization constraints, however, need to be made more restrictive. In the `deadlock-avoidance` class, requests are not granted just because they are available; they are only granted if that will leave the system in a "safe" state (i.e. a state that can not lead to deadlock). The additional synchronization constraints that express deadlock avoidance is given in this restriction:

```
may-lead-to-deadlock(request) prevents access(request)
```

We assume that `may-lead-to-deadlock` is a function that runs the bankers algorithm. Restrictions in the `different-resources` and `resource-administrator` classes are inherited in the `deadlock-avoidance` class. In the resource administrator hierarchy the synchronization constraints become increasingly more restrictive in subclasses. This example illustrates that the restriction-based approach supports incremental modification of synchronization constraints. The `access` method defined in `different-resources` and `deadlock-avoidance` reuse the restrictions associated with the ancestor method. □

The pattern **all-except** is useful when describing synchronization properties that should hold uniformly for all methods ever to be defined in subclasses. Describing such generic synchronization properties will be difficult without the **all-except** pattern since the names of the restricted methods can not be referred to directly. The following example illustrates a case for which the **all-except** pattern comes in handy:

Example 4. Lockability is an example of a generic synchronization property that involves all methods ever to be defined in subclasses. If a subclass inherits the lock property then none of its methods are enabled when the class is locked. An abstract lock class can be described like this:

```
( not locked )    prevents   unlock
( locked )        prevents   all-except unlock
```

An instance variable, `locked`, is declared. This variable is set to `true` by the `lock` method and to `false` by the `unlock` method. The restrictions specify that an object can not be unlocked unles it is locked, and that `unlock` is the only method that can be called when the object is locked. □

Some class hierarchies may require **selective inheritance** of restrictions, i.e. the ability *not* to inherit certain restrictions from certain classes. Selective inheritance is necessary if superclass methods are redefined from scratch in subclasses or if a class has multiple superclasses (multiple inheritance). However, selective inheritance is not necessary for all inheritance mechanisms and therefore, it is not part of our notation. Still, the principles of our framework are general and flexible enough for selective inheritance to be smoothly integrated.

One possible way to describe selective inheritance is for a class to set up **iron-curtains** that limit the inheritance of certain restrictions from certain superclasses. Iron-curtains could point out restrictions not to be inherited in the form invocation patterns. One possible semantics would be to have all invocations that match a given pattern *not* be restricted by the synchronization constraints applying in a specified superclass. We will not discuss iron-curtains any further. We only want to point out that our framework does not preclude integration of mechanisms that allow selective inheritance of superclass constraints.

3.4 Aggregation of Restrictions

In this section we argue that the predicate **disabled** introduced in Section 3.2, should be available as a function that may be called explicitly.

Calling **disabled** as part of a condition makes it possible to describe a restriction as an aggregate of other restrictions. Aggregation of restrictions is important since it may cater for reuse by composition as a supplement to reuse by inheritance. Also, aggregation more directly allows the expression of relations between the restrictions associated with different methods. The following two examples illustrate the use of aggregation.

Example 5. Consider Example 2 with a `get-2` method in a subclass of a bounded buffer. With aggregation we can directly express the fact that `get-2` is disabled whenever `get` is disabled:

```
( size ≤ 1 ) or ( disabled get) prevents get-2
```

In that way no exact knowledge is needed in the subclass about the restrictions that apply to `get`. Additional restrictions may be added to the bounded buffer's `get` method and these additional restrictions will also apply to the subclass and `get-2` without changing the subclass description. In general, aggregation allows subclasses to maintain the consistency notion of a superclass without knowing its exact nature. □

Example 6. Suppose another method, `access-specific`, is added to one of the resource administrator classes in Example 3. The parameters to `access-specific` contain a reference to a resource and `access-specific` makes it possible for clients to access that specific resource. An invocation of `access-specific` is only allowed when the requested resource is available. A motivation for having two access methods is a scenario in which some clients can use any resource while other clients must use one specific resource. The restrictions associated with `access-specific` can be conveniently described in terms of the restrictions associated with `access`.

```
( not available(request)) or ( disabled access(request))
prevents access-specific(request)
```

The function `available` computes the availability of a given resource. □

4 Concluding Remarks and Related Work

In summary, we found that inheritance of synchronization constraints requires incremental restriction of synchronization constraints. Our framework for describing synchronization constraints exhibits the following basic principles:

- Inheritance of synchronization constraints described by pattern-based application of restrictions.
- Aggregation of synchronization constraints by explicit inquiry about the legality of given invocations.

We believe these two principles constitute an appropriate basis for specification of synchronization constraints in concurrent object-oriented languages. Part of the framework has been implemented in an experimental Actor language [7]. The view that synchronization constraints get increasingly more restrictive in subclasses has proven useful in practice.

It may seem that our notion of increasingly restrictive synchronization constraints may violate substitutability properties of subclasses. In [15] and [14], for example, it is argued that subclasses should contain at least the "input offers" of superclasses, implying that synchronization constraints get increasingly *less* restrictive in subclasses. The motivation for this approach is subtype substitutability in the following sense: the type of a server object denotes contractual obligations between clients and that server. A server type guarantees clients that certain input-offers will be made by the server. The guarantees given by a servers type must be maintained by subtypes of the server. Otherwise, subtypes would not be substitutable for supertypes.

With our approach to subclassing and the above typing scheme, subclasses would not be subtypes. However, it is debatable, in general, whether subclasses always need to be subtypes. Furthermore, it is an open question whether synchronization constraints should have any significance for the types of objects. First of all, a type violation yields an error whereas a "violation" of the synchronization constraints most often causes an invocation to be delayed. Secondly, it is not obvious that the type of a server can give communication guarantees to any *one* client in a system where multiple clients may share the server. Interference between clients in the form of interleaving of service requests may undermine such type-guarantees.

Few object-oriented concurrent languages support incremental modification of synchronization constraints. In Rosette [20], synchronization constraints are described as enabled sets: data structures that denote the currently enabled methods. Subclasses can incrementally *add* methods to enabled sets defined in superclasses. As opposed to our framework, synchronization constraints in Rosette get *less* restrictive in subclasses. Beta [9] and the language proposed by Thomsen [19] contain explicit input actions similar to the accept statements of Ada [21]. Subclasses can incrementally *add* more input actions, which again means that synchronization constraints get *less* restrictive in subclasses. The author does not know of any other framework that supports the view that synchronization constraints get increasingly more restrictive in subclasses.

To keep the framework as simple and general as possible, the issue of serializing method invocations was deliberately ignored. It should be noted, however, that restrictions in concurrent invocations can easily be expressed within the framework. For example, conditions in restrictions could refer to synchronization counters [4].

Acknowledgements

The author is sponsored by a research fellowship from the Natural Science Faculty of Århus University in Denmark and generous support from the Danish Research Academy.

The research described in this paper was carried out at the University of Illinois Open Systems Laboratory (OSL). The work at OSL is supported by grants from the Office of Naval Research (ONR contract number N00014-90-J-1899), Digital Equipment Corporation, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195).

The author is grateful to Gul Agha, Chris Houck, Ole Agesen, Christian Callsen, Rune Dahl, Nayeem Islam and Daniel Sturman for inspiring discussions about synchronization constraints and careful reading of this paper.

References

1. P. America and F. van der Linden. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *OOPSLA '90 Proceedings*, 1990.
2. G. Bracha and W. Cook. Mixin-based Inheritance. In *OOPSLA '90 Proceedings*, 1990.

3. N. Carriero, D. Gelernter, and J. Leichter. Distributed Data Structures in Linda. In *POPL '86 Proceedings*, 1986.
4. D. Decouchant, P. Le Dot, M. Rivelli, C. Roisin, and X. Rousset de Pina. A Synchronization Mechanism for an Object Oriented Distributed System. In *Eleventh International Conference on Distributed Computing Systems*. IEEE, 1991.
5. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
6. J. E. Grass and R. H. Campbell. Mediators: A Synchronization Mechanism. In *Sixth International Conference on Distributed Computing Systems*. IEEE, 1986.
7. C. Houck. Run-Time Support for Distributed Actor Programs. Master's thesis, University of Illinois at Urbana-Champaign, 1992. Forthcoming.
8. J. L. Knudsen. Name Collision in Multiple Classification Hierarchies. In *ECOOP'88 European Conference on Object-Oriented Programming*. Springer Verlag, 1988.
9. B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. The BETA Programming Language. In B. D. Schriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.
10. B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Classification of Actions or Inheritance Also for Methods. In *ECOOP'87 European Conference on Object-Oriented Programming*. Springer Verlag, 1987.
11. S. Matsuoka, K. Wakita, and A. Yonezawa. Analysis of Inheritance Anomaly in Concurrent Object-Oriented Languages. ECOOP/OOPSLA'90 Workshop on Object-Based Concurrent Systems, August 1990.
12. S. Matsuoka, K. Wakita, and A. Yonezawa. Synchronization Constraints With Inheritance: What is Not Possible — So What is? Technical Report 10, Department of Information Science, the University of Tokyo, 1990.
13. C. Neusius. Synchronizing Actions. In *ECOOP'91 European Conference on Object-Oriented Programming*. Springer Verlag, 1991.
14. O. Nierstrasz and M. Papathomas. Towards a Type Theory for Active Objects. In D. Tsichritzis, editor, *Object Management*. University of Geneva, 1990.
15. O. Nierstrasz and M. Papathomas. Viewing Objects as Patterns of Communicating Agents. In *OOPSLA '90 Proceedings*, 1990.
16. E. Shibayama. Reuse of Concurrent Object Descriptions. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*. Springer Verlag, 1991. LNCS 491.
17. A. Silberschatz, J. Peterson, and P. Galvin. *Operating Systems Concepts*. Addison-Wesley, third edition, 1991.
18. B. Stroustrup. An Overview of C++. *Sigplan Notices*, October 1986.
19. K. S. Thomsen. Inheritance on Processes, Exemplified on Distributed Termination Detection. *International Journal of Parallel Programming*, 16(1), February 1987.
20. C. Tomlinson and V. Singh. Inheritance and Synchronization with Enabled-Sets. In *OOPSLA '89 Proceedings*, 1989.
21. United States Department of Defense. *Reference Manual for the Ada Language*, draft, revised mil-std 1815 edition, july 1982.
22. J. van den Bos and C. Laffra. PROCOL, a Concurrent Object-Oriented Language with Protocols Delegation and Constraints. *Acta Informatica*, 28:511 – 538, 1991.

This article was processed using the LaTeX macro package with LLNCS style