

Towards Reusable Synchronisation for Object-Oriented Languages

David Holmes, James Noble, John Potter
Microsoft Research Institute and Department of Computing
School of Mathematics, Physics, Computing and Electronics
Macquarie University, Sydney, Australia
email: dholmes, kjx, potter@mri.mq.edu.au

May 22, 1998

Abstract

The integration of concurrency and object-orientation has been the subject of much research for well over a decade. This integration has been problematic due to the conflicts that can arise between synchronisation and inheritance which limit the reuse potential of concurrent objects. The aim of our research is to provide a means for achieving flexible and reusable synchronisation within existing object-oriented languages. We have identified the different aspects of synchronisation which need to be addressed and have developed the underlying “synchronisation rings” model for creating synchronised objects. Current work focuses on developing a synchronisation ‘aspect’ language to allow easier use of the synchronisation rings model and to attain the reuse goal we desire.

1. Introduction

Objects and concurrency started off together back in the late sixties with Simula [Dah70], but tended to go their own way for a while. Concurrent objects resurged in the eighties with the view that objects equate to processes - the so called “active object” model [Yon87] in which a thread within the object accepts messages and then invokes methods. Conversely, passive objects do not have their own thread of control, instead methods are executed in threads that originate from other objects. Concurrent object-oriented languages exist which support purely active models [Agh87, Lie87], purely passive [Baq95, Gos96, McH94, Mit95, Str91] or a hybrid combination [Act92, Ame87, Car90, Jal93, Kar92, Loh92, Mey96, Nie92, Yok87]. The pure active models were based on the notion of ACTORS [Agh86], but having a process (or thread) per object is simply not practical on existing hardware systems, thus things tended to move towards hybrid approaches where passive objects are contained within active ones. On the other hand object languages, such as Smalltalk [Gol83], C++ [Str91] (with library support) and Java™ [Gos96], adopted the view that objects are inherently passive and instead provided means for creating new threads of control.

The early languages did not concern themselves with the reuse issues of synchronisation at all [Ame87, Lie87], and even today languages like Smalltalk and Java take no special steps to enable reuse of synchronisation code. But as more concurrent object-oriented languages were proposed with various features for creating and controlling concurrent interactions, so problems with reuse began to surface. Eventually these reuse issues were studied in more detail and the term “inheritance anomaly” was coined to describe them [Mat90]. Since then most language proposals have been promoted as “solving” the inheritance anomaly [Baq95, Ber94, Fer95, Fro92, Lop94, Mat93, McH94, Mit95]. Unfortunately such proposals generally suffer from one, or both, of two flaws: first they present a very simple concurrency model which offers little in the way of flexibility or control - things which are needed for practical concurrent programming; and second while they may have addressed some of the issues raised by the “inheritance anomaly” they fail to address the real issue of achieving reuse.

Our goal has not been to develop a new research language for concurrent object-oriented programming but to develop techniques for effectively applying and reusing synchronisation within existing languages such as Java or C++. As such we are addressing that form of concurrent programming generally termed “multi-threading” rather than parallel programming. By examining recent work in this area [McH94, Mit95, Lop97] we observed the trend towards “separation of concerns” and how that led into the notion of “Aspect-Oriented-Programming” [Kic96]. In [Hol97] we claimed that although synchronisation is often considered a single monolithic aspect, it is in fact composed of a number of aspects and we identified three primary aspects of synchronisation. Since then we have identified two additional secondary aspects [Hol98]. Our underlying model for synchronisation is based on message interception and redirection and uses the metaphor of “synchronisation rings” [Hol97] which encapsulate a core, functional object, thus providing a synchronised object. Within that model each ring enforces a constraint, which itself belongs to one of the primary aspects of synchronisation. Although our model provides the level of control and flexibility that we desired, it is difficult to use directly. The next phase in the development of our system is to define a synchronisation ‘aspect’ language, which will allow the easy specification of synchronised objects whilst maintaining the flexibility we require and supporting the original goal of effective reuse.

Section two outlines the different aspects of synchronisation that we have identified whilst section 3 describes the synchronisation rings model. Section 4 discusses the requirements for a synchronisation aspect language and in section 5 we look at the current status of the project.

2. Aspects of Synchronisation

In object-oriented systems all interactions between objects occur via the sending and receiving of messages. The receiving of messages leads to method invocations which can lead to further messages; thus the execution of a program forms a trace of messages between objects. In a concurrent object-oriented system it is possible for more than one message send to be in progress at the same time, with the same target receiver. To prevent these concurrent message sends from interfering requires synchronisation, either of the senders or of the receiver. In object-oriented systems we can view synchronisation of concurrent objects as the control of message acceptance: what messages will be accepted, when and from whom. If we examine the potential reasons for not being able to accept a message we find three primary aspects of synchronisation [Hol97]:

Exclusion	A message can not be accepted as it would cause interference with another message already being processed
State	A message can not be accepted because the internal state of the object does not allow the message to be processed correctly
Coordination	A message can not be accepted because the sender of the message does not hold the right to send the message to this object at this time

There are in addition two secondary aspects of synchronisation which must also be considered when dealing with the primary aspects:

- If a primary constraints is not met what **response** should be given by the object?
- If messages are waiting in what order should they be **scheduled** for processing?

Each aspect of synchronisation manifests itself as a constraint on the acceptance of a message, which must be enforced for the correct processing of the message. Following the design principle of ‘separation of concerns’, by focussing on these aspects individually we get a clearer perspective on why different forms of synchronisation are needed within our programs. When enforcing any given type of constraint may require a sophisticated policy, by treating each aspect individually we potentially achieve a finer granularity of reuse of those policies (and their implementations) and also achieve greater control over the variations in policy possible for a given object.

2.1 Exclusion

Exclusion constraints define which messages are allowed to be accepted concurrently without causing interference. Exclusion constraints never depend on the logical state of an object, that is the values of its fields, but solely on the ‘execution’ state of the object, that is which messages are currently active.

2.2 State

State constraints restrict when a message can be accepted and processed due to the internal state of the object. State constraints involve the availability of internal resources. For example, if a bounded buffer is full then a `put()` message can not be accepted and if the buffer is empty a `get()` message can not be accepted; in the former case we require a ‘space’ resource and in the latter an ‘item’ resource.

State constraints are abstract properties of an object that exist in both sequential and concurrent environments. Concurrency does not cause state constraints to come into existence, but gives us alternative ways to respond to them because a message can be delayed until the receiver is in the correct state to accept it.

A state constraint may depend not only on the current state of an object, but also on the past state of the object, the past execution state of the object (e.g. which message was last accepted), or ultimately the state of the environment (e.g. the day or time).

2.3 Coordination

Coordination constraints restrict when a message may be accepted based on the sender of the message. They are termed coordination constraints because they involve the coordination of multiple messages to multiple objects. A typical use of a coordination constraint is to enforce atomicity over the processing of a sequence of independent messages. For example, consider a system which includes two bounded buffers and a client that wishes to transfer the contents of one buffer into the other in the same order. The exclusion and state constraints enforced by each buffer do not help the client in achieving its goal if other clients can also send messages to the buffers, thus a coordination constraint is required to ensure that messages from other clients get rejected whilst the atomic transfer is in progress.

2.4 Response

For each of the primary aspects of synchronisation discussed we need to define how an object responds if the constraints are not met when a message is received. There are a wide variety of possible responses to finding a constraint not satisfied [Lea96] but the most common responses are:

- balking** returning immediately with an indicator that the operation did not proceed because a constraint was not met
- blocking** waiting indefinitely until the constraint is met
- timed-wait** wait up to a specified maximum time for the constraint to be met. If the constraint is not met then some indication that the operation timed out should be given

These can all be viewed as forms of timed-waits with balking indicating a time-out of zero and blocking indicating a time-out of infinity.

Our sequential heritage has predisposed us to expect, and provide, balking responses in many practical, state constrained situations. For example, even though written to be concurrency ‘safe’, the collection libraries of Java [Sun98] and the Booch Components [Boo93] both simply return null for messages that could not be accepted due to the state of the collection. Yet developers of concurrent object-oriented languages have generally only provided blocking responses to state constraints. To provide the greatest flexibility a programmer should be able to dictate what response is to be given, even on a per-call basis.

2.5 Scheduling

Scheduling constraints determine the order in which multiple messages blocked at an object, should be processed. These scheduling constraints influence synchronisation in two ways. Firstly, we sometimes extend an existing constraint to take into account whether or not some other type of message is waiting. For example, a readers/writer policy may use the constraints on a reader that there are no active writers and no waiting writers, thus giving a priority writer protocol; without the second constraint we would have a priority reader protocol. Secondly, scheduling constraints influence the order in which newly enabled messages are processed. Commonly we choose a first-in-first-out ordering as it is considered fair. Sometimes though we have specific priorities assigned to different message senders, in which case a priority ordering is more appropriate.

In theory scheduling constraints can be arbitrary and based on a range of different criteria. Bloom's criteria [Blo79] include such things as the sender of the message, the time the message was sent/received, the nature of the message and the parameters to the message. The ability to provide a variety of scheduling constraints has been defined by some as a measure of the expressiveness of a concurrent object-oriented language [McH94, Mit94], though in practice they seem to be infrequently used.

3. "Synchronisation Rings" Model

We have developed a model where a synchronised object is a combination of separate functional and synchronisation behaviours. Each aspect of synchronisation required by the object is provided by one or more synchronisation rings which surround a core, functional object¹ (Figure 1). A ring encapsulates state and provides a number of ports through which messages can interact with that state. Each ring provides a particular synchronisation behaviour by processing each method invocation (message) and determining if that invocation can proceed, based on the port used and the current state of the ring. These behaviours enforce the synchronisation constraints of the object.

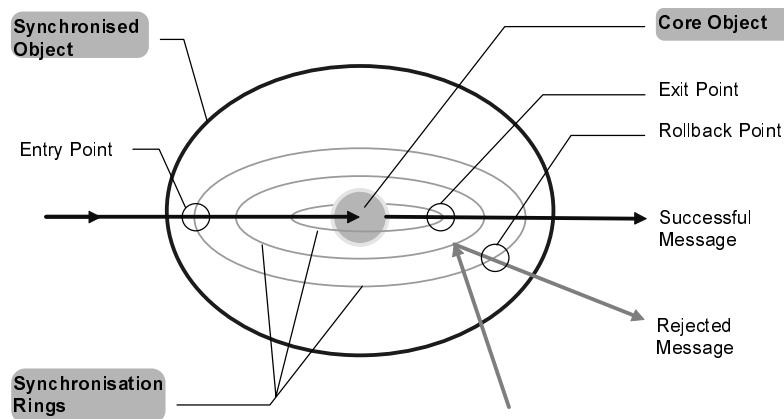


Figure 1 The Synchronisation Rings Model

3.1 Message processing

In the synchronisation rings model, method invocation is viewed as the passing of a message to the synchronised object. This message is directed to each synchronisation ring to get the appropriate synchronisation behaviour and to the core object to execute the desired functional operation. In the general case synchronisation requires that a message be processed

¹ The core object may itself be an aggregation of functional objects or synchronised objects.

both before and after it is handled by functional code. This pre- and post-processing leads to a model of messages entering and exiting rings that surround a functional core.

A message must pass through each ring, indicating that the constraints for processing that message are met. As each synchronisation behaviour can implement a number of constraints, each ring can have a number of access ports. These consist of an entry, exit and rollback point. The entry point of a port ensures that a particular constraint is met before the message can pass through and will usually change some local state to influence the passage of future messages. Once a message has passed through all rings it can be processed by the core object. Once processed by the core object the message continues on and passes through each ring again, this time through exit points. The exit point belongs to the same port as the entry point through which the message was originally processed and complements that entry point by completing the change of state that represents the end of a successful operation.

If a message can not pass through a ring it is rejected by the port. A rejected message bounces off the ‘closed’ ring and returns back through each of the rings it has entered so far. Each ring processes a rejected message at the rollback point of the port at which the message entered. The rollback point must undo any state changes performed by the entry point and thus allow further messages to be processed. This rollback protocol allows different synchronisation behaviours to be composed without causing local deadlock.

To support the blocking of rejected messages, there is a queue associated with each port. Messages may be removed from queues when other messages complete or rollback, or when their own time-out has elapsed. The conditions for acceptance of a message at a port may include the state of the ring and the state of any of the queues used by the ports of that ring.

Although messages conceptually pass through each ring they need not be actively processed by each ring, so an implementation can optimise for rings that have no effect on particular messages. Nor is it necessary for messages to map to an operation on the core object: some messages will manipulate synchronisation state only. For example, enabling or disabling a bounded buffer doesn’t have any effect on the buffer itself. A ‘pure synchronisation’ object has no core object but only synchronisation rings.

4. A Synchronisation Aspect Language

Programming the synchronisation rings model directly, is a laborious and tedious task, and not one in which the benefits of reuse are apparent. To simplify how a programmer uses the synchronisation rings model we are developing a synchronisation ‘aspect’ language which can be used to specify how to build a synchronised object

Defining such an aspect language is a non trivial task as the synchronisation problem itself is non-trivial. As well as obtaining ‘functional’ correctness in our synchronisation schemes we must also address the practical issues that affect the performance of real systems. Additionally we must address the needs of programmers who require synchronisation mechanisms which are both flexible and which can be controlled in specific ways. Finally we must produce a language which facilitates re-use. Combining all of these into a single language is not simple and thus what follows is not a complete description of our language, but a discussion of the issues we must address in developing our language with some possible syntax for those areas we have already fleshed out in more detail..

4.1 Exclusion

Our goals for expressing exclusion constraints are three-fold:

1. We require the ability to define complex exclusion protocols involving fine grained granularity of locking, using mutual exclusion sets, or reader/writer sets, as well as the more simple protocols of reader/writer and mutual exclusion.
2. We want to give the programmer the ability to specify a simple exclusion protocol at the time an object is created: reader/writer, mutual exclusion or no exclusion.

3. We must be able to easily define the exclusion constraints for new methods in a derived class and to change the exclusion constraint for an existing method.

An additional requirement that must be considered when providing synchronisation for languages such as C++ and Java is how to synchronise access to class, or static, data. This issue is usually ignored in other systems.

There are two basic approaches for defining exclusion constraints: we can specify the requirements of each method, or we can group similar methods together by specifying a particular policy. In the first approach we simply specify what the exclusion constraints on a method are: for example, that `put()` excludes `put()` and `get()`. In the second approach we define the policy used by the various methods: for example: `policy mutex(put, get)`. These two approaches differ in how well they support our goals.

Using the first approach we capture the true requirements of each method in a simple form and it is easy for a new method to specify which of the existing methods it excludes. This gives us the exclusion matrix for the object from which we can identify various possible policies for enforcing the matrix. Unfortunately this does not provide us with a means to specify which policy we require. Further, although mutual exclusion is trivial to enforce the mapping from the exclusion matrix to a reader/writer policy is not necessarily one-to-one. Finally care must be taken in how the exclusion matrix is formed when we change the exclusion constraint of an existing method in a derived class - how do we tell if an exclusion constraint replaces or enhances an existing one?

On the other hand, if we are going to specify policies directly then we must have a terminology for expressing those policies. We could simply indicate a binding between a method and a lock and then allow the runtime implementation to determine the nature of the lock and thus the policy enforced. However, this requires a semantic understanding of the different types of policies if we are to convert complex policies into simpler forms. Alternatively we could use the policy to create the exclusion matrix but that still leaves a problem if there is more than one mapping to the simpler policy. Adding a new method should be relatively straight forward as should changing an existing constraint - but again we need a way to distinguish between modifying a policy and replacing it.

4.2 State

The enforcement of state constraints is a complex matter. The range of possible state constraints seems unlimited and thus we would seem to need to be able to express any programmable statement within our aspect language. That would seem impractical given that programming languages already exist that do a very good job in forming such arbitrary expressions. Instead the synchronisation rings model captures specific constraints within rings and the ports on those rings define how a method interacts with a given constraint. Although the exact details of a constraint will depend on the object and methods being defined, the form of those constraints are often similar. Synchronisation rings are designed to capture the generic form of a constraint and we have two main types of ring for dealing with state constraints: counters and event rings.

Counters can be used to model both single-bounded and double-bounded counts and the ports on a counter allow for a range of interactions with the counter. The increment port requires that the count be below its upper bound and at completion the count has been incremented; whilst the decrement port checks the lower bound and decrements the count. Other ports allow 'reservations' in which we require that a count can be increased/decreased and we prevent other methods from changing the count such that it can no longer be increased/decreased, but at the end of the operation we leave the count unchanged. Such a reservation is needed for an operation like `peek()` which returns the item at the head of the buffer. Here's an example of a possible synchronisation specification for a bounded buffer (ignoring exclusion requirements):

```

synchronisation BoundedBuffer {
  BoundedBuffer(int capacity); // constructor information
  local count Counter(0,capacity);

  void put(Object item){
    notFull: uses count.increment; // a labelled clause
  }
  Object get(){
    notEmpty: uses count.decrement;
  }
  Object peek(){
    notEmpty: uses count.reserveDecrement;
  }
}

```

The `uses` clause binds a method to a port on a ring, whilst the naming of those clauses allows for the individual overriding of clauses in a derived class.

Events are simple rings which reflect whether or not an event has occurred. We define sufficient ports to reflect what state the event must be in before the port can accept a message and what affect that message has on the port. Thus a port can either require that the event be set, clear or doesn't care and it can either set the event, clear it or leave it unchanged. Such events are useful for many things including the classic history-state sensitivity example of `gget()` in the bounded buffer - a `gget()` is just like a `get()` but can only occur if the last operation was a `get()`

```

synchronisation GgetBoundedBuffer like BoundedBuffer{
  local lastOpWasGet Event(); // add a new event ring

  Object gget() like super.get {
    lastOp: uses lastOpWasGet.requireAndClear;
  }
  Object get() {
    lastOp: uses lastOpWasGet.set;
  }
  void put(Object item){
    lastOp: uses lastOpWasGet.clear;
  }
}

```

Here we see how one synchronisation specification can utilise another by declaring itself like another. We add the new `gget()` method indicating that it's constraints are like those of the original `get()` method and then add the new binding. For the existing methods we simply add the new binding showing how they affect the event.

However the approach adopted by the synchronisation rings model only works well for state constraints where the affect of each method on a constraint is known a priori. For example if our buffer includes a `purge()` operation which discards entries that are out-dated then we do not know what the final size of the buffer will be until after the operation has complete. This requires the ability to directly interact with a ring, passing it information, at the completion of the operation.

4.3 Coordination

As coordination usually involves multiple objects, coordination requirements need to be described at the object trying to perform the coordinated activity. This requires that the objects which are the targets of the coordinated activity either support coordination directly or that the runtime system allows coordination to be enforced on any object. The former approach has the problem that potential for coordination must be identified a priori, whilst the latter generally requires a homogenous object system. As component architectures become more pervasive we can see a need to coordinate between objects in different object systems (eg.. C++ objects, Java objects, CORBA components, COM components). However, the coordination problem is so complex that it is itself the subject of much research [Fro93, Her97]. Solving such a general problem is beyond the scope of what we can achieve but what we can do is allow simple support for coordination within the synchronisation rings model.

To support coordination we will allow the ability to acquire exclusivity rights both at the object level and the method level. Thus an operation to atomically swap the contents of one system buffer with another could have a synchronisation specification as follows:

```
void swap( exclusive Buffer a, exclusive Buffer b) {}
```

While an operation to atomically transfer the contents of one buffer to another may only need exclusive use of specific methods on the destination buffer:

```
void transfer(exclusive Buffer from, Buffer to){  
    exclusive to.put();  
}
```

If the transfer operation is built into the buffer itself then we could have:

```
void transfer(Buffer to){  
    exclusive this, to.put();  
}
```

The actual means for providing exclusive use has not yet been established..

4.4 Response

The ability to request a particular response to an un-met constraint lies with the client. As such requests are done on a per-call basis, mapping such requests to method level synchronisation specifications is not practical. There are two basic approaches for providing different responses: either annotate the call site with information concerning the required response, or provide different methods which respond in different ways. Both approaches are complicated by the possibility of requiring different responses to different constraints.

Adding annotations to the call-site is perhaps the easiest way of expressing such requirements, but of course is not amenable to use in an existing language such as C++ or Java. Defining different methods with different responses seems impractical given the possible permutations of responses to different constraints.

At present we propose only limited support in this area. Given that a practical need for balking or timed-out exclusion constraints rarely arises, we will only support blocking in this context. Likewise coordination constraints will only exhibit a blocking response. For state constraints we propose to generate three methods for each method that has a state constraint - one for each response. The general rules for doing this are as follows:

- The normal method signature represents the indefinite blocking response
- The normal signature with an extra numerical argument indicates a time limited wait
- For methods with void return types a new method with a boolean return type, and with the name augmented with 'try', will be created to give a balking response - returning true if the operation completed and false if the operation bailed

Complications arise with overloaded methods, where it is not possible to distinguish between actual method arguments and the timeout argument, and methods that return values. For methods that do return values we propose the use of special flag values to indicate success or failure where possible. In some cases this approach will simply not be possible. Details on this still need to be worked out.

4.5 Scheduling

As for state constraints the scheduling requirements that are possible using Bloom's criteria are far too varied to support directly within the synchronisation aspect language. Instead we propose a programmatic approach whereby scheduling criteria is encapsulated within an appropriate comparator object which is then used by the underlying synchronisation rings implementation. We can provide simple comparators for things like FIFO ordering and thread-priority ordering, whilst more complex comparators can be created by the programmer. However, the exact means for establishing this have not yet been determined.

5. Current Status and Future Work

Whilst the synchronisation rings model and its current implementation have evolved over the past year, the interface to this model, using a synchronisation aspect language, is still in its infancy. The ideas outlined in the previous section have yet to be implemented and obviously many of the issues need further examination. We plan to develop our language as far as possible in the available time and to integrate it with the synchronisation rings implementation. Meanwhile we are interested in finding means for ‘weaving’ aspects at finer granularity than the method level as it seems that a synchronisation aspect language would benefit from this.

References

- [Act92] Acton D. and Neufeld G., “*Controlling Concurrent Access to Objects in the Raven System*”, International Workshop on Object-orientation in Operating Systems Principles, 1992
- [Agh86] Agha G., “*ACTORS: A Model of Concurrent Computation in Distributed Systems*”, MIT Press, 1986
- [Agh87] Agha G. and Hewitt C., “*Concurrent Programming Using Actors*”, in “Object Oriented Concurrent Programming”, A. Yonezawa and M. Tokoro (Eds.), MIT Press, pp37-54, 1987
- [Ame87] America P., “*POOL-T: A Parallel Object-Oriented Language*”, in “Object Oriented Concurrent Programming”, A. Yonezawa and M. Tokoro (Eds.), MIT Press, pp199-220, 1987
- [Baq95] Baquero C., Oliveira R. and Moura F. “*Integration of Concurrency Control in a Language with Subtyping and Subclassing*”, In Proceedings of the USENIX Conference on Object-Oriented Technology and Systems (COOTS), 1995
- [Ber94] Bergmans L.M.J., “*Composing Concurrent Objects: Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs*”, Ph.D. dissertation, University of Twente, Netherlands, 1994
- [Blo79] Bloom T., “*Evaluating Synchronisation Mechanisms*”, Seventh International Symposium on Operating Systems Principles, pp24-32, 1979
- [Boo93] Booch G. and Vilot M., “*Simplifying the Booch Components*”, in “C++ Report”, Vol. 5 No. 5, pp. 40-52, June 1993
- [Car90] Caromel D., “*Concurrency: An Object-Oriented Approach*”, Proceedings of TOOLS-2, pp183-197, 1990
- [Dah70] Dahl O.J. , Myrhaug B. and Nygaard K., “*SIMULA Common Base Language*”, Norwegian Computing Centre S-22, Oslo, Norway, 1970
- [Fer95] Ferenczi S., “*Guarded Methods vs. Inheritance Anomaly*”, SIGPLAN Notices 1995
- [Fro92] Frolund S., “*Inheritance of Synchronisation Constraints in Concurrent Object-Oriented Programming Languages*”, ECOOP’92, 1992
- [Fro93] Frolund S. and Agha G., “*A Language Framework for Multi-Object Coordination*”, ECOOP’93, 1993
- [Gol83] Goldberg A. and Robson D., “*SMALLTALK-80 - The Language and its Implementation*”, Addison-Wesley, 1993
- [Gos96] Gosling J., Joy B. and Steele G., “*The Java™ Language Specification*”, Addison-Wesley, ISBN 0-201-663451-1, 1996
- [Her97] Hernandez J., Papatomas M., Murillo J. and Sanchez F., “*Coordinating Concurrent Objects: How to deal with the coordination aspect*”, position paper in the Aspect Oriented Programming Workshop at the European Conference on Object- Oriented Programming, June 1997
- [Hol97] Holmes D., Noble J. and Potter J., “*Aspects of Synchronisation*”, Proceedings of TOOLS Pacific ’97, IEEE Press, 1998
- [Hol98] Holmes D., Noble J. and Potter J., “*Effective Synchronisation of Concurrent Objects: Laying the Inheritance Anomaly to Rest*”, (submitted for publication) April, 1998
- [Jal93] Jalloul G. and Potter J., “*A Separate Proposal for Eiffel*”, Proceedings of TOOLS Pacific ’93, Prentice Hall, 1993
- [Kar92] Karaorman M. and Bruno J., “*A Concurrency Mechanism for Sequential Eiffel*”, Proceedings of the Eighth International Conference on Technology of Object Oriented Languages and Systems (TOOLS8), pp63-77, 1992

- [Kic96] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J and Irwin J. , “*Aspect-Oriented Programming*”, Position paper, Xerox PARC, Aspect Oriented Programming Project”, 1996
- [Lea96] Lea D., “*Concurrent Programming in Java™: Design Principles and Patterns*”, Addison-Wesley, ISBN 0-201-69581-2, 1996
- [Lie87] Lieberman H., “*Concurrent Object-Oriented Programming in Act 1*”, in “Object Oriented Concurrent Programming”, A. Yonezawa and M. Tokoro (Eds.), MIT Press, pp9-36, 1987
- [Löh92] Löhr K., “*Concurrency Annotations Improve Reusability*”, Proceedings of the Eighth International Conference on Technology of Object Oriented Languages and Systems (TOOLS8), pp53-62, 1992
- [Lop94] Lopes C. V. and Lieberherr K. J., “*Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications*”, Proceedings of the Eighth European Conference on Object Oriented Programming (ECOOP 94), Springer-Verlag, pp 81-99, 1994
- [Lop97] Lopes C. V., “*D: A Language Framework for Distributed Computing*”, Ph.D. Dissertation, College of Computer Science, Northeastern University, Boston, 1997
- [Mat90] Matsuoka S., Wakita K. and Yonezawa A., “*Synchronisation Constraints with Inheritance: What is not possible? - so what is?*”, Technical Report 10, Dept. of Information Science, University of Tokyo, 1990
- [McH94] McHale C., “*Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*”, Ph.D. Dissertation, Department of Computer Science, University of Dublin, Trinity College, October 1994
- [Mey96] Meyer B., “*Object-Oriented Software Construction*”, 2nd Edition, Prentice-Hall, 1996
- [Mit95] Mitchell S. E., “*TAO - A Model for the Integration of Concurrency and Synchronisation in Object Oriented Programming*”, Ph.D. Dissertation, Dept. of Computer Science, York University, UK, 1995
- [Nie92] Nierstrasz O., “*A Tour of Hybrid: A Language for Programming with Active Objects*”, in “Advances in Object-Oriented Software Engineering”, Prentice Hall, pp167-182, 1992
- [Str91] Stroustrup B., “*The C++ Programming Language*”, Second Edition, Addison-Wesley, ISBN 0-201-53992-6, 1991
- [Sun98] Sun Microsystems, “*The Java Collections Framework*”, on-line API documentation, “<http://java.sun.com/products/jdk/1.2/docs/guide/collections/>”, 1998
- [Yok87] Yokote Y. and Tokoro M., “*Concurrent Programming in ConcurrentSmalltalk*”, in “Object Oriented Concurrent Programming”, A. Yonezawa and M. Tokoro (Eds.), MIT Press, pp129-158, 1987
- [Yon87] Yonezawa A. and Tokoro M., “*Object-Oriented Concurrent Programming: An Introduction*”, in “Object Oriented Concurrent Programming”, A. Yonezawa and M. Tokoro (Eds.), MIT Press, pp1-8, 1987