Synchronizing Actions

Christian Neusius
Universität des Saarlandes
FB14 Informatik
W-6600 Saarbrücken, Germany
email: pool@cs.uni-sb.de

Abstract

A model of concurrency control, *synchronizing actions*, is presented specifically designed for concurrent object-oriented programming languages (COOPL). A current research problem in COOPL is the conflict arising from contradictory objectives related to concurrency and encapsulation. Synchronizing Actions presents a solution for this kind of problem. The model supports extension and reuse of a system, the major goals of object-oriented programming, i.e. it provides guidelines for the design of concrete synchronization mechanisms such that they do not interfere with inheritance. Synchronizing actions are a design frame rather than a specific realization. The model is not restricted to a specific model of concurrency, as for example serialization of method executions. We will show the suitability of the model by giving a specific synchronization mechanism based on this design frame.

1. Introduction

We will apply the term *object-oriented* in the sense of Wegner [Weg87], i.e. a language is object-oriented if it provides at least objects, classes and class-based inheritance. Our attention is focused on concurrent object-oriented programming languages (COOPL) that support the concurrency model of *active* objects as the concurrently executing entities. Note that some languages provide active and passive objects (passive objects are private to active objects); this approach allows balancing the degree of concurrency.

Nierstrasz and Papathomas state in [Nie90] that "none of the existing approaches (that combine inheritance and concurrency) has yet succeeded in resolving basic conflicts between concurrency mechanisms and encapsulation that is needed for the safe use and reuse of object-oriented code". Synchronizing Actions are a step towards the solution of this conflict; the model provides a clearer separation of concurrency control and implementation details than all existing approaches. Closely related to this problem are the interferences between concurrency control and inheritance discussed by Decouchant [Dec89] and Kafura and Lee [Kaf89]. Kafura and Lee state that decentralized interface control is a necessity when aiming at reusability and extensibility of object-oriented applications. In the model Synchronizing Actions a guideline is deferred from the

This work was supported by the Deutsche Forschungsgemeinschaft, SFB124, TP D3.

valuable approaches in ACT++ [Kaf89] and Rosette [Tom89]. This guideline alleviates the design of synchronization mechanisms that meet the requirements of the Object-Oriented Programming Paradigm.

In section 2 we evaluate synchronization mechanisms that are based on the decentralized interface control approach, and show how they interfere with encapsulation. The languages ACT++ [Kaf89] and Rosette [Tom89] provide synchronization mechanisms that do not interfere with inheritance. However, their synchronization mechanisms impose a restriction to the concurrency model, namely serialization of method executions per object. After shortly discussing the usefulness of internal concurrency, the model Synchronizing Actions is presented that refines the model of decentralized interface control from section 2. The model describes a design frame which allows different realizations of concurrency control. As an example we will show a concrete synchronization mechanism using this design frame. Finally some open problems and future research directions are mentioned.

2. Existing Synchronization Mechanisms

The design of COOPL providing active objects led to the development of several new synchronization mechanisms. The commonality of these mechanisms is the concept of *interface control* which will be explained below. First, we will briefly review these synchronization mechanisms and then discuss their strengths and shortcoming in combining inheritance and concurrency.

2.1 Language Classification

There are currently several languages combining Concurrency and Object-Oriented Programming. These emerged from two distinct motivations:

(a) Adding Concurrency to an OOPL

The extension is motivated by the claim that an OOPL should be good for modeling the real world [Mad88]. Thus an OOPL should support concurrency in a better way than it is in OOPLs as Simula-67 [Dah66] or Smalltalk [Gol83]. The new approach is thus to transform objects into active entities as in Beta [Kri87] or Concurrent Smalltalk [Yok86].

(b) Adding Object-Oriented paradigms to a Concurrent Object-based language.

Here the extension is motivated by the aim "Structured Programming" and support of code reuse and extensibility. Thus parallel programs become applicable to support the development of similar systems.

These different points of view influenced the language design, and particularly the design of synchronization mechanisms. Amongst the first COOPL was Concurrent Smalltalk (CST) [Yok86]. CST evolved from the object-oriented language Smalltalk [Gol83], and thus belongs to category (a). Like other languages of this category, CST provides *self reference* as an essential design criterion. The model of internal concurrency within an object was chosen since self reference interferes with the serialization of method executions. The consequences of this design decision will be discussed later.

Exponents of category (b) are ACT++ [Kaf89] and Rosette [Tom89]. They were developed on the basis of *actors* [Agh87]. The *actor* model emphasizes as model of concurrency the serialization of method executions within an actor (i.e. an active object). While adding classes and class-based inheritance, self reference was not considered in the language design. Some notes to the essence of self reference are given later.

2.2 Decentralized Interface Control

Interference of synchronization and inheritance was noted by Decouchant [Dec89], Kafura and Lee [Kaf89]. When synchronization is not properly separated from the specification of methods, the extension of code by adding a subclass may force the change of code within the superclass. This, however, contradicts the design rules of OOP. Note that a superclass may have several subclasses, and a change within a superclass will thus withdraw severe consequences. Kafura and Lee give a raw classification of synchronization mechanisms and state that only *decentralized interface control* can be combined with inheritance without restriction of reuse and extensibility.

The task of an object's *interface control* is to decide at a given moment if a message pending within the object's mail-queue may enter the object and thus may be processed or not. Once started, a method execution is no longer explicitly synchronized by the concurrency control. It may be involved in synchronization only when calling another method and waiting for the result. The decision of the concurrency control depends on the content of the mail-queue and on the state of the object's concurrency control. This state defines the set of messages that currently can enter the object; it is represented by a set of data. We will call this set of data the Interface Control Space (ICS) of the object.

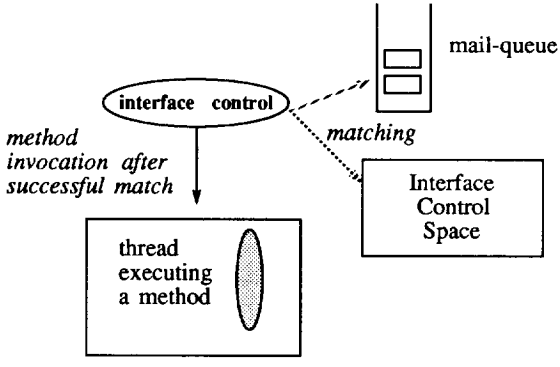

Figure 1. Execution Model *interface control*

When giving a detailed model of interface control we must consider the underlying model of concurrency. We compare in the following two models, *serialization* of method executions, and *internal concurrency* within an object.

2.2.1 Serialization of Method Executions

If an object has at most one active thread that has access to the ICS and data shared between methods, and this thread is involved in the execution of one method then method executions are serialized. During this execution the object is in the state **locked**, i.e. no further method can be invoked. As soon as the thread terminates, the object becomes unlocked. Then the concurrency control within the interface will be invoked. It matches the current state of the Interface Control Space with the mail-queue that contains the messages sent to the object. When a message matches with the ICS the corresponding method will be invoked, and the object becomes locked again. Otherwise the object remains delayed as long as no further message arrives and triggers the invocation of concurrency control.

Examples

ACT++ [Kaf89] provides *behavior abstractions* as the mechanism of concurrency control. Behavior abstractions are sets of method names. A method that returns control (i.e. unlocks the object) specifies by the choice of a behavior abstraction the set of methods that can be invoked next. A behavior abstraction represents an entity that can be redefined within subclasses. The matching process for *behavior abstractions* is a simple test, if there is a message in the mail-box such that the related method is in the state **open**.

Enabled-sets in **Rosette** [Tom89] have some similarities to the mechanism of *behavior abstractions* (as e.g. being redefinable entities of the ICS) but they provide a higher flexibility. The method name (and eventually some of the actual parameter values) contained in a queued message have to match an enabled-set in the Interface Control Space. Tomlinson and Singh call this mechanism matching by content, and compare it with the tuple space model in Linda [Car86].

Note that there may be more than one active thread within an actor in ACT++ and Rosette. This is because a thread may continue running after execution of the replacement behavior of the Concurrency Control. After that it is no more capable of influencing the Concurrency Control or to access data accessible by several methods. There is at most one thread that has the ability to change the Concurrency Control.

2.2.2 Internal Concurrency

An object may have several active threads executing method invocations. The object is thus always in a state unlocked. Nevertheless, the concurrency control ensures, that arriving messages will be delayed until their turn has come. Whenever a message arrives or a thread terminates, the concurrency control will be triggered. Once triggered the concurrency control may create several threads *at once* before terminating.

Examples

Guide [Dec89] provides boolean expressions attached to methods, the activation conditions. A method may be executed when its activation condition returns *true*. Activation conditions use amongst other parameters synchronization counters (introduced independently by Gerber [Ger77]

and Robert and Verjus [Rob77]). For example, a counter *started*(m) is automatically increased by the system immediately before the creation of a thread executing method m, and *started*(m)-*completed*(m) delivers the number of threads currently executing method m. Since these counters are independent from method specification they are the key mechanism that supports the model of internal concurrency. Using only the counters, however, would not be flexible enough. Consequently, an activation condition uses instance variables that represent the internal state of the object. A programmer has to consider the consistent change and use of these variables within the method specification.

The synchronization mechanism of **DCST** [Nak89] is a combination of Method Relations and Method Guards. A Method Relation defines the constraint under which methods are not allowed to run concurrently. A Method Guard is a boolean expression attached to a method that evaluates instance variables. Obviously only the method relations are independent from method specification. This independency is necessary to support the model of internal concurrency since otherwise the consistent change of the instance variables used in Method Guards cannot be achieved.

2.3 Discussion

The source of trouble when combining concurrency control and inheritance is the dependency of the method specifications and the synchronization protocol of an object. The problems so far identified with this dependency concern (a) defining a new method in a subclass may interfere with the synchronization protocol of many superclass methods and (b) encapsulation is weakened. Below, we will discuss if and how the presented synchronization mechanisms solve these problems.

2.3.1 Interference of Concurrency Control and Inheritance

The languages ACT++ and Rosette support reuse and extension of systems in the way that changes of the concurrency control can be specified locally in a class without the need of changing code in other classes or rewriting inherited methods. The applied synchronization mechanisms separate the Interface Control Space from the normal set of data of the object (i.e. the instance variables), and build up the ICS by independent, **redefinable** units. The applied units are defined as sets that are easily changeable (in Rosette) or redefinable (in ACT++) by adding (subtracting) elements to (from) it. Each set represents a specific state of the object's Concurrency Control.

Guide and DCST do not sufficiently support reuse and extension of systems. In Guide, the main shortcoming is the strong attachment of method names and synchronization counters. Adding a method means also adding new synchronization counters. These synchronization counters unfortunately can only be reflected in the inherited activation conditions by redefining the attached inherited methods. Note also that a redefined method gets its own, new set of counters which additionally complicates the synchronization protocol.
In DCST changes of the synchronization protocol due to newly defined methods in a subclass cannot be reflected within the methods of the superclass or their related Method Guards . Thus one may have to redefine all inherited methods that are involved in the synchronization protocol.

2.3.2 Weak Encapsulation

All synchronization mechanisms mentioned above have one specific shortcoming in common, the conflict between the concurrency control and encapsulation. The data involved in the concurrency control (ICS) are accessible to the concurrency control of the subclasses. Since these data are used and even changed within methods encapsulation in the sense of Snyder [Sny87] is weakened. The impact of this weakness is shown below.

The interface control as defined in the above languages captures only a small part of the tasks of the concurrency control. These tasks consist of

(a) the decision of which methods may be executed or not and

(b) the change of concurrency constraints that are evaluated within this decision.

The task (a) may be characterized as the *matching phase*, and is equivalent to the definition of the interface control. The task (b) can be characterized as the *state transition phase* of the concurrency control. None of the synchronization mechanisms above has a clear separation of this *state transition phase* and the specification of implementation details. In Guide, the change of synchronization counters is separated from method specification, but the change of instance variables used in the activation conditions has to be specified within the methods. The same holds for DCST, where the Method Relations are separated from implementation details, but not the Method Guards. In Rosette and ACT++, the state transition phase is given by the computation of which set will override the old set in the ICS. This computation is specified within the methods.

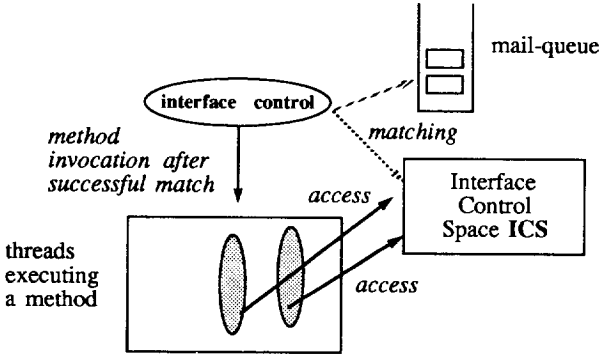

Figure 2. Weakening of Encapsulation.

Figure 2 outlines the violation of encapsulation. This *violation* complicates the understanding of the synchronization protocol since the changes of instance variables belonging to the ICS are hidden in the implementation. Additionally, the approach is particularly critical for the model of

internal concurrency. Note that the change of the ICS itself requires mutual exclusion of methods that access the same data of the ICS. This can force the cautious programmer to introduce unnecessary serializations in an application.

Consider, for example, a bounded buffer providing put- and get-operations, where internal concurrency is supported. As soon as the thread executing a put is started, the ICS must switch from the acceptance of put requests to the delay of put requests. Another critical situation occurs when a variable counting the content of the buffer is used within the Interface Control, but changed within the put- and get-operations (as it would be the case in DCST). Then a put and a get operation running concurrently will both access this counter, thus violating mutual exclusion. The complexity of Concurrency Control dramatically increases when internal concurrency is supported.

3. Why Internal Concurrency ?

We will illuminate first the importance of self reference for object-oriented programming. Wegner and Zdonik [Weg88] emphasize self reference as an essential property of object-oriented programming.

> *"In a world without self-reference, inheritance reduces to invocation and inheritance hierarchies are simply tree-structured resource-sharing mechanisms."*

Self reference allows the definition of small sized redefinable units; the code to be rewritten when extending or reusing a system can be held minimal. The problem with *self* in COOPLs is the interference with the serialization of method executions [Yok86]. In the design of Concurrent Smalltalk [Yok86, Yok87] self is reduced to a local procedure call within the atomic, serializing objects. This is a severe restriction, and it is inapplicable when we apply decentralized interface control. While not entering the object by its interface, method invocations by self are excluded from concurrency control.

On the other hand, internal concurrency combined with self reference has the following advantage. A method may be invoked and then be executed up to that moment, where synchronization inevitably must take place. Here it will enter this synchronizing phase by calling self. Then it is delayed as long as the event occurs that allows the processing of the *synchronizing method*, and the calling thread regains control. An example will be shown in section 5.

Finally, a discussion at [WS89], p. 10 about the concurrency models in concurrent object-based languages may be cited. The supporters of the internal concurrency model argue that there should be concurrency within objects for reasons of performance and because there are objects in the real world that exhibit internal concurrency. The main argument against this model is the complexity of synchronization mechanisms (concurrent access of shared variables).

4. Synchronizing Actions

The dependency of the method specifications and the synchronization protocol of an object is the weak point that makes reuse and extensibility so difficult. In comparison to the traditional

critical section approach the strength of the decentralized interface control lies in the (partial) separation of concurrency control and method specification. Nevertheless the model underlying the existing COOPLs still conflicts with the paradigm of encapsulation. The design frame *synchronizing actions* separates, besides the "matching phase", also the "state transition phase" of the concurrency control from the method specification. The data of the ICS are encapsulated by the concurrency control since only operations of the concurrency control have access to them. The instance variables are encapsulated by the methods accessing them. Synchronizing Actions can thus be seen as a necessary refinement of the idea "decentralized interface control". This refinement eases reuse and extension of code.

4.1 The Synchronization Model

The new synchronization model is characterized as follows. The synchronization of a method is defined by its *matching_rule*, a set of operations to be executed before the invocation of the method (*pre_action* specification), and a set of operations to be executed after termination of the method (*post_action* specification).

- The *matching-rule* is a boolean expression that evaluates the content of the mail-queue (messages and eventually parameters) and the state of Concurrency Control.
- The *pre_action* and the *post_action* specify the change of the concurrency control that becomes necessary by the invocation or termination of a method call, respectively. They consist of a sequence of operations working solely on the Interface control space. Pre_action and post_action are executed as atomic actions.

The set of matching-rules, pre_actions and post_actions of the methods are the only specifications that can access the ICS, i.e. they evaluate or change the state of the interface control. We will show how the mutual exclusion is guaranteed for the execution of them in the new execution model.

4.2 New Execution Model

The execution of the matching-rules, pre-actions and post-actions of the methods are serialized to achieve mutual exclusion. The Concurrency Control consists of one single thread, and calls synchronously the pre-actions and post-actions (see Figure 3). These may thus be seen as local procedures.

The Concurrency Control is triggered by an event "terminating method execution" or by an event "arrival of a new message". It works then as follows. When the concurrency control was started by the event "method execution terminated" it will execute the appropriate state transition defined within the *post_action* specification of the method. After that it will match if pending messages can be accepted for invocation. After a positive match for a method m the state transition - caused by the invocation of m - will be performed as defined in pre_action (m). Then a thread will be created for the execution of method m. The cycle "match, do pre_action(m), invoke(m)" is repeated until no more requests match for invocation.

By strengthening encapsulation, synchronizing actions also help in decreasing the conflict between concurrency control and inheritance. Nevertheless, the following guideline (learned from ACT++ and Rosette) must be considered in the design of a concrete synchronization mechanism.

Guideline.

When we add or redefine a method in a subclass - while extending the system - we must be able to reflect this within the concurrency control defined so far in the superclass(es). The amount of redefined code must be minimal. This can be achieved by defining the entities within the ICS as being independent and (eventually) redefinable.

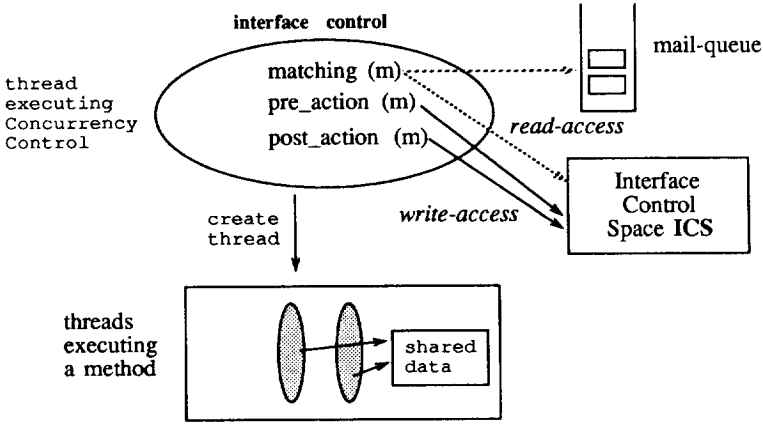

Figure 3. Separating Methods from ICS.

Independency of the entities of the ICS from the method specifications is one characteristic property of our model *synchronizing actions*. The second property of an entity in the ICS "to be redefinable" is sometimes necessary, for example when method names are used within the ICS. In the synchronization protocol of a bounded buffer, for example, all methods that try to get an element from an empty buffer should be delayed including those methods that may be added later within a subclass. Thus the entities in ACT++ and Rosette representing such a state "empty buffer" in the ICS are redefinable sets of method names.

As we will see in the next section an entity of the ICS does not necessarily have to be redefinable (see counter N in example *bounded buffer* where the condition N=0 represents the state "empty buffer").

5. An Exemplary Synchronization Mechanism

We will present the *bounded_buffer* example also given in [Kaf89] and in [Tom89], so a comparison with these contributions is possible. Nevertheless, the degree of internal concurrency

for the example is small. The synchronization mechanism defined below should be seen as an exemplary *ad hoc* solution. Detailed investigations on flexible and easily usable synchronization mechanisms are one of our current research topics.

We define the synchronization mechanism as follows. Instead of a method's state **open** or **closed** as in ACT++ we introduce a *lock()* counter for each method that counts the number of lock- and unlock-operations on a method. A method **m** is locked by another method **m2** (to guarantee mutual exclusion) by an *exclude* statement within a pre_action specification; lock(**m**) is thus increased. When method **m2** terminates, the *exclude* statement is implicitly reversed by an unlock of the method **m**, i.e. lock(**m**) is decreased. The *exclude* operation is thus temporarily bound to the method execution. Note that the **implicit** matching rule for a method m is (*lock*(m)=0); when this condition is not fulfilled, further matching is unnecessary.

The arguments of *exclude* statements are sets of methods that may be redefined within subclasses. We will call these sets *behavior abstractions* analogously to ACT++. Besides these sets we allow the specification of variables within the Interface Control Space. These variables are accessible only by matching operations, pre_actions and post_actions. Only simple assignment operations to the variables of the ICS can be specified within the pre- and post-part.

5.1 Example "Bounded Buffer"

A bounded buffer acts as a FIFO queue where the operation put adds a new element to the tail of the queue, and the get operation removes an element from the head of the queue and returns it to the caller as result. To ensure mutual exclusion of operations on the same data, we define the sets op_on_head = { get } and op_on_tail = { put }. When a put is to be executed, an "**exclude** op_on_tail;" is called in the pre_action of put(). Thus, no other operations may act concurrently on this part of the shared data. As a further control mechanism we have to delay a put() operation when the buffer is full, and delay the get when the buffer is empty. The states full or empty of the buffer are captured by a counter N of the queue's content. It is evaluated in the matching rule, and changed within the post_actions of put() and get(). Note that the example is written in such a way that one put() and one get() may act concurrently because they access disjunct parts of the shared data. The example is shown in figure 4.

Extended Bounded Buffer

Now, we will show the extension of bounded_buffer by the method get_rear() as presented by [Kaf89] and [Tom89]. Note that put() and get_rear() act on the same position of the bounded_buffer and thus cannot act in parallel. The same will happen with get() and get_rear() if there is exactly one element queued in the bounded buffer. To give thus an easy solution, the execution of get_rear() requires the exclusion of another get_rear() as well as put() and get(). A more complex solution where get() and get_rear() may act concurrently (i.e. the content of the buffer is greater than 1) is not given. The redefined sets *op_on_head* and *op_on_tail* must be extended by adding get_rear(). The complete example is given in figure 5.

Note that the **exclude** mechanism used above could arbitrarily be replaced by simply using counters *op_on_head* and so on within the matching condition and the pre- and post-actions.

```
class bounded_buffer;

private:
    SIZE = 64;
    int in=0, out=0, buf[SIZE];
Concurrency_control:
    int N = 0;    // counts queued elements
    behavior abstraction
              op_on_head = { get }
              op_on_tail = { put }
public:
    method put (int elem);
         matching          (N<SIZE);
         pre       {    exclude op_on_tail; }
         action { increase in and add elem on tail of buf}
         post      {    N++;}
    method get ():int;
         matching          (N>0);
         pre       {    exclude op_on_head; }
         action { return element from head of buf and increase out }
         post      {    N--; }
end bounded_buffer;
```

Figure 4. *bounded_buffer.*

```
class extended_bounded_buffer inherits bounded_buffer;

Concurrency_control:
    behavior abstraction
         // redefinitions :
             op_on_head = { get, get_rear }
             op_on_tail = { put, get_rear }
         // new exclusion set for get_rear()
             op_on_head_or_tail = { put, get, get_rear };
public:
    method get_rear ():int;
         matching          (N>0);
         pre       {    exclude op_on_head_or_tail; }
         action { return element from tail and decrease in }
         post      {    N--; }
end extended_bounded_buffer;
```

Figure 5. *extended_bounded_buffer.*

5.2 Example "await_an_event"

In the following example a method *sync_on_event*() will execute up to that moment where it has to wait for the occurrence of a specific event. The occurrence of the event will be propagated to the object by calling a method *event_occurred*(). The occurrence of the event is represented in the ICS by a variable event_count. The example shows the use of self in the synchronization protocol. It also illustrates the separate execution of concurrency control issues (matching, pre- and post-part in the method specification) and the method operating on the instance variables of the object (action part in the method specification).

Note that the method event_occurred() does not provide an action part. The purpose of the "invocation" of method event_occurred() is to define a state transition of the concurrency control. The interface control does not have to create a thread executing the action part of the method. Other optimizations may be considered in a concrete language design as e.g. the efficient evaluation of the matching operations, or allowing a synchronous call of methods that provide no action part (see method await_event() in the example).

```
class Event_Example;
concurrency_control:
        int event_count = 0;
protected:   // i.e. visible only to subclasses
   method await_event() : bool;
        matching (event_count>0);
        pre { }
        action { return true }
        post { event_count-- }    // consume one event

public:
   method event_occurred();
        matching (true);
        // since action_part is empty, omitt pre- and action-part
        post { event_count ++ }
   method sync_on_event ();
        matching (true);
        pre { }
        action { ...
            // now synchronize on occurrence of event
            ok:= self!await_event();
            ... }
        post { };

end Event_Example;
```

Figure 6. *await_an_event* example.

6. Concluding Remarks

Kafura and Lee argue that the problem of synchronization mechanisms based on interface control is that defining a new method in a subclass may invalidate many superclass methods [Kaf89]. The approaches in ACT++ and Rosette solve this problem by defining a reflexive synchronization mechanism. The source of trouble of existing synchronization mechanisms in COOPL is the dependency of method specifications and the synchronization protocol of an object. This dependency makes reuse and extensibility so difficult. Changing the synchronization protocol automatically leads to changing the methods. Compared to the traditional *critical section* approach *interface control* as it is understood in the existing approaches is already one step towards a solution of this problem.

The synchronization concept *synchronizing actions* presents a further step towards better solutions by augmenting the independency between the method specifications and the synchronization protocol. Additionally, a basic conflict between encapsulation and concurrency control is solved by introducing two distinct sets of data. The concurrency control exclusively works on the interface control space, whereas the methods exclusively work on the traditional instance variables of the object. The new approach reduces substantially the complexity of the concurrency model supporting internal concurrency within an object compared to existing approaches.

Investigations on flexible synchronization mechanisms obeying the model of *synchronizing actions* are underway. Many fruitful ideas from the existing synchronization mechanisms can be adapted and combined.

In the future research on synchronization in COOPL the development of large applications programmed in COOPLs will be of significance. The experiences made in writing real applications are small compared to concurrent object-based languages as POOL-T [Ame87]. By this way, it must be inspected if the complex model of internal concurrency is appropriate for developing, understanding and maintaining systems. When rejecting this complex model one has to dispense with self reference, or at least this will lead to considerations about a new semantics of self reference.

Similar guidelines like those for the use of the Object-Oriented Programming Technique ([Mey88], [CACM90]) must be developed for COOPL in order to enforce the correct use of synchronization mechanisms. This depends on experience in programming and is thus closely related to writing real applications.

Acknowledgments

I am grateful to the anonymous referees, H. Scheidig, R. Spurk and R. Schäfer for their comments on earlier versions of the paper.

References

[Agh87] Agha, G. and Hewitt, C. Actors : A Conceptual Foundation for Concurrent Object-Oriented Programming. In *Research Directions in Object-Oriented Programming*, ed. B.Shriver and P.Wegner, MIT-Press 1987, pages 49-74.

[Ame87] America, P. POOL-T : A Parallel Object-Oriented Language. In *Object-Oriented Concurrent Programming*, ed. A.Yonezawa and M.Tokoro, MIT-Press 1987, pages 198-220.

[CACM90] Special issue on Object-Oriented Design. *CACM* Vol.33, No.9 (Sept. 1990).

[Car86] Carriero, N., Gelernter, D., and Leichter,J. Distributed Data Structures in Linda. In Proc. of *POPL 13*, ACM, 1986, pp.236-242.

[Dah66] Dahl, O.-J. and Nygaard, K. Simula - An Algol-based Simulation Language. *CACM* 9:9 (Sept.66), pp. 671-678.

[Dec89] Decouchant, D. et al.: A Synchronization Mechanism for Typed Objects in a Distributed System. In [WS89], pages 105-107.

[Gol83] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Ger77] Gerber, A.J.. Process Synchronization by Counter Variables. *ACM Operating Systems Review*, Vol.11 (4), Oct 1977, pp. 6-17.

[Kaf89] Kafura, D.G. and Lee, K.H. Inheritance in Actor Based Concurrent Object-Oriented Languages. In Proc. of *ECOOP'89*. BCS Workshop Series, jul. 1989. Cambridge University Press, pp. 131-145.

[Kri87] Kristensen,B.B., Madsen,O.L., Møller-Pedersen,B. and Nygaard,K. The BETA Programming Language. In *Research Directions in Object-Oriented Programming*, ed. B.Shriver and P.Wegner, MIT-Press 1987, pages 7-48.

[Mad88] Madsen,O.L, and Møller-Pedersen,B. What object-oriented programming may be - and what it does not have to be. In Proc. of *ECOOP'88*, LNCS 322, Springer, 1988, pp. 1-20.

[Mey88] Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Nak89] Nakajima, T. et al.,: Distributed Concurrent Smalltalk. In [WS89], pp. 43-45.

[Nie90] O.Nierstrasz, M.Papathomas. Viewing Objects as Patterns of Communicating Agents. In *OOPSLA ECOOP'90 Conference Proc.*, ed. N.Meyrowitz. Special issue of SIGPLAN Notices 25 (10), Oct. 1990.

[Rob77] Robert, P., Verjus, J.-P. Toward Autonomous Descriptions of Synchronization Modules. *Information Processing 77*, North Holland, 1977, pp. 981-986.

[Sny87] Snyder, A. Inheritance and the Development of Encapsulated Software Components. In *Research Directions in Object-Oriented Programming*, ed. B.Shriver and P.Wegner, MIT-Press 1987, pages 165-188.

[Tom89] Tomlinson, C. and Singh, V. Inheritance and Synchronization with Enabled-Sets. In *OOPSLA'89 Conference Proceedings*, ed. N.Meyrowitz. Special issue of SIGPLAN Notices 24 (10), Oct. 1989.

[Weg87] Wegner, P. Dimensions of Object-Based Language Design. In *OOPSLA'87 Conference Proceedings*, ed. N.Meyrowitz, Oct. 1987. Special issue of SIGPLAN Notices 22 (12), Dec. 1987.

[Weg88] Wegner, P. and Zdonik, S.B. Inheritance as an Incremental Modification Mechanism. In Proc. of *ECOOP'88*, LNCS 322, Springer, 1988, pages 55-77.

[WS89] *ACM SIGPLAN Workshop on Concurrent Object-Based Language Design.* Special issue of SIGPLAN Notices 24 (4), April 1989.

[Yok86] Yokote, Y. and Tokoro, M. The Design and Implementation of Concurrent Smalltalk. In *OOPSLA '86 Conference Proc.*, ed. N.Meyrowitz, Portland, Oregon, Sept. 1986. Special issue of SIGPLAN Notices 21 (11), Nov. 1986.

[Yok87] Yokote, Y. and Tokoro, M. Experience and Evolution of Concurrent Smalltalk. In *OOPSLA'87 Conference Proceedings*, ed. N.Meyrowitz, Oct. 1987. Special issue of SIGPLAN Notices 22 (12), Dec. 1987.