Experience in Classification and Reuse of Synchronization Schemes

Jean-Pierre BRIOT*

Dept. of Information Science
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

tel: +81 (3) 3812-2111 ext. 4116
fax: +81 (3) 5689-4365
e-mail: `briot@is.s.u-tokyo.ac.jp`

Abstract

This paper describes a framework for modeling, classifying, and combining various *synchronization schemes* for object-oriented concurrent programming (OOCP in short). This is achieved by using a general purpose OOCP generic architecture/platform, named Actalk. The architecture is based on the notions of components, parameterization and inheritance. In this paper we focus on the *activity component* of an (active) object in order to express control over acceptance and execution of method invocations. Variations on the parameters (namely virtual methods) of the activity component are used to describe and implement various synchronization schemes. At first, our platform helps in classifying and experimenting with various synchronization schemes. Secondly, it helps in reusing them and building-up on previous expertise in order to derivate enhanced or/and novel schemes. To illustrate this in the paper, we progressively develop various schemes by successive refinements and combinations, starting from two foundational synchronization schemes (namely *enabled sets* and *guards*). Before concluding, we briefly discuss some general architectural issues of generic software platforms by comparing the design of the Actalk architecture to some other similar platforms.

Keywords: concurrency, synchronization, schemes, modeling, platform, generic architecture, components, parameterization, refinement, combination, implementation, Smalltalk-80.

*Also member of LITP, Institut Blaise Pascal, Universités Paris VI & VII - CNRS, 4 place Jussieu, 75252 Paris Cedex 05, France. E-mail: `Jean-Pierre.Briot@litp.ibp.fr`.

1 Introduction

Object-oriented concurrent programming (OOCP in short [OOCP87]) is a very promising methodology for development of parallel, distributed and open applications. It is a generalization of object-oriented programming taking into account multiple activities. The main approach of OOCP is *integrative*, by identifying an object with an autonomous activity (therefore named an *active object*)[1]. This integrative approach also identifies synchronization between objects/activities (named *inter-object synchronization*) at the message passing level. This unification of object with activity, and of message passing with inter-object synchronization, eases the issue of synchronization of concurrent programs.

Meanwhile, in order to express more control over the *availability of services* offered by an object, one may want to express constraints over the possible acceptance of specific patterns of requests/method invocations (e.g., disabling insertion onto a buffer while it is full). Furthermore, if we allow/free an object to process several messages concurrently (this is called *intra-object concurrency*), we need to (re)gain control over concurrent activations of messages in order to preserve object consistency.

We call *synchronization constraints* the specifications of the control/constraints over the way invocations are scheduled and processed by an active object. Various formalisms for synchronization, we will call them *synchronization schemes*, have been proposed to achieve good expressive power, modularity, and efficient enough implementations. Many of them are actual derivations from general concurrent programming [AS83], e.g. *guards*, and have been integrated within the object-oriented concurrent programming framework.

Unfortunately, since the late 80's it has been found that it is not always trivial to reuse such specifications, and *especially* by using inheritance (which is the main standard tool for reuse and specialization of object abstractions). This problem has been coined as the *inheritance anomaly* phenomenon and various proposals have been studied and compared by Satoshi Matsuoka et al. [MY93]. Experience shows that there is no silver bullet, and that abstract formalisms such as *"body" live routine* or *path expressions* proved to be too centralized and therefore hard to reuse. Consequently, almost all currently proposed schemes belong to one of the two following basic families (in other words are derivations of):

- *enabled (methods) sets*: the active object specifies the sets of method patterns which are currently enabled, i.e. that it is willing to accept/serve, see Section 3.1,
- or *guards*: some boolean activation condition is associated to each method, see Section 3.3.

We therefore think that it is useful to allow the designer and the programmer to customize various approaches/schemes, each with its own merits, and to compare and test them along with possible requirements (underlying computation model, granularity of objects and concurrency, emphasis on expressivity versus efficiency, type of applications...). In other words, we are willing to provide a unified framework where various synchronization schemes may be expressed, implemented, compared, refined, combined, and applied to various programming contexts (computation models, languages constructs, programs).

In this paper we describe such a generic software architecture for modeling and classifying various OOCP languages constructs within a single framework/environment. This platform is named Actalk, and its initial design has already been described in [Bri89]. In this paper we describe its application to a constructive taxonomy of various synchronization schemes for active objects. Our approach leads to the following benefits. At first, it makes the classification and the comparison of various synchronization schemes/proposals easier (and more fair). Secondly, it provides opportunity to select and experiment with various schemes for various parts of a whole program/system. Last but not least, this environment helps to design and experiment with new schemes by successive refinements and possible combinations of current ones.

Outline

This paper is organized as follows. Section 2 briefly discusses and describes the Actalk generic architecture to model and classify various OOCP mechanisms. Then in Section 3, we successively

[1] This one-to-one identification between object and activity is the general model of an *active object*. Meanwhile, this does not preclude the existence of passive objects in some OOCP hybrid systems, nor the fact that some kinds of active objects may own multiple sub-activities (*intra-object concurrency* as discussed below).

model various synchronization schemes, through step by step refinement and combination. Section 4 quickly summarizes and evaluates our approach and results, and then compares them to other relevant works before concluding this paper.

2 A Generic Architecture for OOCP Design Alternatives

2.1 Motivations

Object-oriented (concurrent) programming is based on a few simple and generic concepts: (active) objects and message passing. Concepts are strong enough to help at structuring and encapsulating computation modules, and generic enough to encompass various software and hardware architectures. Meanwhile, various alternative programming models, constructs and mechanisms have been proposed in various programming languages and systems. They reflect a wide variation of concerns and domains. Possible variations on programming language design may be grouped into various issues, such as:

- *communication*: is it synchronous or asynchronous, how is a reply conveyed ?, are there priority schemes ?... ;
- *activity*: is acceptance of messages implicit or explicit ?, is there intra-object concurrency ?... ;
- *synchronization/coordination*: how does an active object control message selection and activation ?

This variety of models, and the difficulty to compare them by abstracting from various associated terminologies, syntax and implementational foundations, led us to design a comprehensive and unified modeling platform.

2.2 The Actalk Platform

The design and building of the Actalk[2] project started in 1988 from the needs for a comprehensive workbench for comparing various object-oriented concurrent programming schemes within a single unified framework/environment [Bri89]. Actalk provides a framework to help in analyzing and classifying existing language constructs and mechanisms, in designing new ones through derivation and combination of existing ones, and in testing them with actual programs within a rich programming environment [LBB91].

2.3 Architecture

The architecture of Actalk includes a *kernel* which models basic OOCP semantics (that is active/serialized objects which communicate by asynchronous/unidirectional message passing). The kernel is composed of a set of *kernel component classes*. Each *component* describes a different aspect of an active object, that is: behavior, activity/synchronization, and communication. Each component class is parameterized, that is some of its functionalities (methods) are specifically intended to be redefined/specialized in subclasses in order to model alternative language designs. Therefore these (virtual) methods are named *parameter methods*.

Note that the granularity and balance of the decomposition of the architecture into components and parameter methods is a very sensitive issue. Finer granularity of decomposition brings greater modularity but at the cost of possible complexity, as well as increasing consistency management and efficiency problems. This issue will be discussed in Section 4.2, when comparing our platform to a few other similar systems.

2.4 Component Classes of the Actalk Kernel

The three main kernel component classes are:

- Class `ActiveObject` describes the *behavior* of the active object, that is the inner passive object which ultimately computes the messages. Application classes of active objects are defined as subclasses of class `ActiveObject`. The language designer may also implement specific programming language constructs in some abstract subclasses (for instance subclass

[2] *Actalk* stands for *act*ive objects, or *act*ors, in Small*talk*-80.

`Abcl1Object` implements the ABCL/1 language construct `wait-for` to explicit wait for some message pattern [OOCP87, pages 55–89]).

- Class `Activity` describes the *internal activity* of the active object. This class provides autonomy for the active object. It also defines the way method invocations are selected, scheduled and computed. Consequently, its subclasses may describe various synchronization schemes.
- Class `Address` describes the *address* (mailbox) of an active object, that is the identifier of an active object where messages will be sent. This class defines the way message transmissions will be interpreted. Its subclasses may implement various types of communication.

This decomposition allows the independent modeling of various dimensions of active objects. One can decouple the actual program from a given communication model, from a given model of activity (e.g., with intra-object concurrency), and from a given synchronization scheme[3]. Meanwhile, combination of arbitrary components may lead to inconsistencies. Therefore the Actalk platform includes some simple compatibility specification and verification mechanism to help keeping safeguards [Bri94].

Note that the kernel actually provides two more kernel component classes: class `MailBox` which represents the message queue, and class `Invocation` which represents a method invocation (by default the message itself, as instance of standard Smalltalk-80 class `Message`). They are not considered as prime components because their parameterization is minimal. Meanwhile their functionality may be extended by subclassing them as for the three main component classes. This proves to be useful when modeling complex protocols for message buffering, e.g. with priorities, or/and for invocation management, e.g. time stamp (as in Section 3.6).

2.5 Relations between Components

To define a class of active objects, the programmer should define it as inheriting from class `ActiveObject`. After creating an active object behavior as its instance, creation and initialization of the associated components (activity, address and mailbox) take place transparently. Default classes for associated components are expressed by specific parameter methods (e.g., parameter method `addressClass` specifying the associated activity component class, see in Table 1).

The minimal functionality of an active object, and the relations/interactions between its components, may be summarized by following the main steps of a message, from its reception to its processing (see Figure 1):

- the address *receives* a message, triggering parameter method `receiveMessage:` of class `Address` which enqueues the message in the mailbox ;
- *independently*, the activity *selects* the next message to be processed (parameter method `nextMessage` of class `Activity`) ;
- and accepts it (parameter method `acceptMessage:` of class `Activity`), ultimately delegating its processing (parameter method `performMessage:` of class `Activity`) to the behavior.

2.6 The Activity Kernel Component Class and its Parameterization

In this paper we focus on the *activity* component classes (i.e., subclasses of kernel component class `Activity`) in order to implement various synchronization schemes. Class `Activity` defines two instance variables: `address` to reference the associated address component, and `bself` to reference the associated behavior component, that is the "inner self" of the active object. Parameter methods of class `Activity` are summarized in Table 1.

2.7 Generic Event Methods

Another important characteristic of the Actalk architecture is the existence of *generic event methods*. These parameter methods are associated to the three following events: message (invocation)

[3] Note that in the architecture, the activity/synchronization component is explicitly and structurally distinct from the behavior/program component. This enforces independence between program code/data and synchronization code/data as advocated in [MBW+94]. This also eases comparison of various synchronization schemes on actual examples by just changing the synchronization class while keeping the same behavior/program.

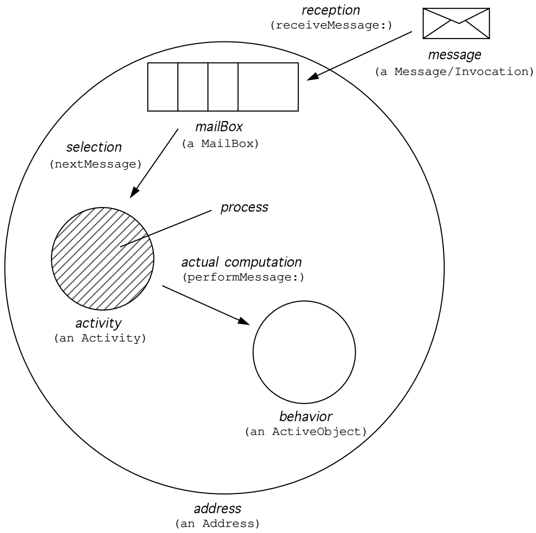

Figure 1: Components of an Actalk active object.

method selector	*parameter*	*default value*	*example of redefinition*
`initialize`	*initialization*	none	initialize synchronization counters (see Section 3.4)
`start`	*start the activity (process)*	start the process created by `createProcess`	start a second activity process specific to ABCL/1 express mode messages
`createProcess`	*create the activity process*	create a process computing `body`	provide a handle to the process (useful for termination control)
`body`	*specification of the activity*	serially accept successive messages	accept a single message (e.g., Actors behavior activity)
`nextMessage`	*next message to be accepted*	return and remove first message from the message queue	return and remove the first message whose selector is enabled (see Section 3.1)
`acceptMessage:`	*accept and compute a message*	`performMessage:`	spawn an inner thread to compute the message (intra-object concurrency class `ConcurrentActivity`)
`performMessage:`	*perform a message*	delegate actual perform to the behavior	also return the value to the implicit reply destination
`addressClass`	*default address component class*	`Address`	`PoolAddress` (associated to class `PoolActivity`)
`invocationClassFor:`	*invocation class*	`Message`	`WithSenderInvocation` (includes the sender)
`eventReceive:/` `Accept:/Complete:`	*message reception/ acceptance/completion generic events*	none	after completion, compute the next enabled set of selectors (see Section 3.1)

Table 1: Parameter methods of component class `Activity`.

reception/acceptance/completion, and are respectively named: `eventReceive:/Accept:/Complete:`. (They take the current message as their argument.)

The user may redefine them to attach actions to a given class of active objects, e.g.: tracing activities, stepping computation, controlling global scheduling of activities... Generic event methods may also be used by the designer for modeling language specificities (e.g., computing post actions for the POOL language [OOCP87, pages 199–220]). They are also very useful for managing intra-object synchronization. Subclass `SynchroConcurrentActivity` specializes them to ensure their atomicity (mutual exclusion), as will be seen in Section 3.4.

2.8 Experiments and Applications

Various *extensions* of the Actalk kernel have been defined as subclasses of one or more of the kernel component classes. These various component subclasses simulate various:

- *language models and constructs*: e.g., Actors concept of behavior replacement [OOCP87, pages 37–53], ABCL/1 explicit wait for a message pattern [OOCP87, pages 55–89], POOL concept of body and post actions construct [OOCP87, pages 199–220]... ;
- *communication models*: e.g., ABCL/1 three types (synchronous, asynchronous, future) and two modes (normal, express) of message transmission, implicit reply mechanism (for a good integration within underlying standard Smalltalk-80 method execution model)... ;
- and *synchronization schemes*: the focus of this paper. Part of the corresponding taxonomy is detailed in Section 3. Other schemes include: explicit message acceptance (as in POOL and Eiffel//), method suspension (as in Portable ConcurrentSmalltalk), reflective framework (as in OCore)...

With its libraries and taxonomies of various OOCP languages characteristics, the Actalk testbed provides a comprehensive view of design alternatives and mechanisms. Actalk has been used as a teaching and experiment foundation by various people, notably in a graduate course led by Jean Bézivin at University of Nantes where student projects developed many experiments. Actalk has also being used as a foundation to simulate software engineering process models [KG84], and to build various multi-agent systems [BFS91], currently being applied to domains such as: natural language processing, knowledge acquisition [BH94], and telecommunications networks supervision.

3 Modeling Synchronization Schemes

We will now model/implement various synchronization schemes and show the expressiveness of our platform. As noted in the introduction, *enabled sets* and *guards* are the two major synchronization schemes foundations. We will use successive refinements and combinations of these two schemes to produce increasingly expressive (and complex) synchronization schemes. Our point is actually in showing how we may easily enhance and customize various synchronization schemes along various requirements while building-up on previous expertise. Figure 2 summarizes the hierarchy of synchronization schemes/classes which will be developed in this section.

Note that we sometimes combine two synchronization schemes into a new one (e.g., class `CountersActivity` described in Section 3.4), thus inheriting from more than one class. As there is currently no multiple inheritance mechanism in Smalltalk-80, we unfortunately must choose a *single* superclass (*solid arrow* in Figure 2) and recopy variables and methods from the other one(s) (*dashed arrow*). These copied methods won't be shown in the definitions given in the paper.

3.1 A First Basic Scheme: Enabled Sets/Abstract States

The first synchronization scheme that we implement is the *enabled sets* synchronization scheme. This scheme is very good at expressing *services availability* constraints, i.e. how an active object is willing to accept certain method invocations. In many cases, parameters are not significant in the decision. One may then introduce a further level of abstraction in enabling (or disabling) method patterns (i.e., selectors) grouped into *enabled methods sets*.

Abstract states are abstractions/names to which enabled sets will be assigned. (In the canonical example of the bounded buffer, where `put:` requests should be disabled while the buffer is full, this will be expressed through the abstract state `full` having only {`get`} as its corresponding enabled set.) After selecting and computing the current method invocation, a state transition function will

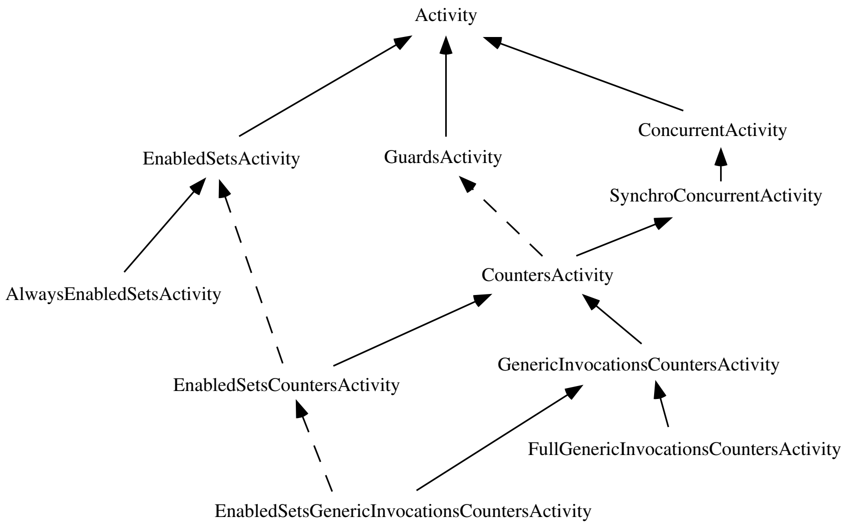

Figure 2: Hierarchy of synchronization schemes/classes developed.

compute the next abstract state (leading to the next enabled set). Enabled sets are computed through set operations such as union `+`, intersection `&`, and difference `-`.

Class `EnabledSetsActivity` implements this scheme in the following way. It defines an instance variable `enabledSelectors` to hold the current enabled set. An enabled set of selectors is implemented as a Smalltalk array (e.g., `#(get put:)`)[4]. An abstract state is represented by a method returning an enabled set. Implementation only redefines three of the parameter methods: `initialize`, `nextMessage`, and generic event method `eventComplete:`.

```
Activity subclass: #EnabledSetsActivity
   instanceVariableNames: 'enabledSelectors '

"Parameter method for initialization of the activity."
initialize
   super initialize.
   "Initialize the initial set of enabled selectors."
   enabledSelectors := self perform: self initialAbstractState

"Return first message from the mailbox belonging to the current enabled set."
nextMessage
   ^self mailBox firstMessageWithCondition:
        [:message | enabledSelectors includes: message selector]

"Generic event method associated to the completion of the computation of a message."
eventComplete: aMessage
   super eventComplete: aMessage.
   "Abstract state transition: computation of the next set of enabled selectors."
   enabledSelectors := self perform: (self nextAbstractStateAfter: aMessage selector)
```

Note that method definitions reference/call two undefined methods: `initialAbstractState` (to specify the initial abstract state) and `nextAbstractStateAfter:` (to compute the next abstract state). These two *virtual* methods are not defined at this abstract level, but must be in a specific (concrete) synchronization subclass (e.g., in next example). Note that the code above is a concise but *complete* implementation.

Example: Synchronization of a Bounded Buffer

The enabled sets/abstract states synchronization scheme is very good at expressing services availability constraints based on the state of the (active) object. In the canonical example of the

[4] `#` is the quotation character in Smalltalk, both for symbols (e.g., `#empty`) and for arrays (e.g., enabled set `#(put:)`).

bounded buffer, there will be three abstract states (note that abstract state **partial** may be defined as the union of **empty** and **full**):

abstract state	*enabled set*
`empty`	`#(put:)`
`partial`	`#(get put:)`
`full`	`#(get)`

The activity/synchronization of the bounded buffer is specified in class **BufferEnabledSetsActivity**, defined as a subclass of class **EnabledSetsActivity**.

The implementation of the associated bounded buffer behavior/program (trivial and here out of scope) is not shown in this paper. We assume that it implements methods **put:** and **get**. We also assume that the bounded buffer behavior implements two predicate methods **isEmpty** and **isFull** and accessor method **maxSize** (to consult the maximum size). These latter methods won't be declared in the external interface of the active object. They are for internal use only, by calling the behavior "inner self" referenced by instance variable **bself** (defined in class **Activity**, see Section 2.6).

```
EnabledSetsActivity subclass: #BufferEnabledSetsActivity
   instanceVariableNames: ''

"Abstract states."
empty
   ^#(put:)

full
   ^#(get)

partial
"Defined as the union (+) of the abstract states empty and full."
   ^(self empty) + (self full)

"The initial abstract state when creating an instance."
initialAbstractState
   ^#empty

"Abstract state transition: computing the next abstract state."
nextAbstractStateAfter: selector
   ^bself isEmpty
       ifTrue: [#empty]
       ifFalse: [bself isFull
                    ifTrue:  [#full]
                    ifFalse: [#partial]]
```

3.2 First Refinement: Considering Always Enabled Methods

Although this is not the focus of this paper, we now briefly mention one example of "inheritance anomaly" phenomenon, and the way we can easily refine the initial synchronization scheme described above in order to address that particular problem[5].

Suppose for example that we define a subclass of the bounded buffer defined above as specifying a new method named **numberOfElements**. This method should always be available, thus enabled. The problem is that we therefore need to add it to all existing enabled sets (or at least to "basic states" **empty** and **full**), thus forcing some amount of redefinition. This may be done semi-automatically by specifying a generic set of always enabled methods. This is easily implemented as the following refinement (subclass) of class **EnabledSetsActivity**:

[5] This is by no means a complete solution to the complete set of "inheritance anomaly" problems, which is *not* the focus of this paper.

```
EnabledSetsActivity subclass: #AlwaysEnabledSetsActivity
   instanceVariableNames: ''

initialize
   "Add the set of the always enabled selectors to the initial enabled set."
   super initialize.
   enabledSelectors := enabledSelectors + self always

eventComplete: aMessage
   "Add the set of the always enabled selectors to the next enabled set."
   super eventComplete: aMessage.
   enabledSelectors := enabledSelectors + self always

"Abstract states."
always
   "Return the set of the always enabled selectors."
   "By default this set is empty."
   ^#()
```

The new buffer activity class (`BufferAlwaysEnabledSetsActivity`) is defined as for previous buffer activity class (`BufferEnabledSetsActivity`), *plus* the specification of the "always enabled" method `numberOfElements`:

```
AlwaysEnabledSetsActivity subclass: #BufferAlwaysEnabledSetsActivity
   instanceVariableNames: ''

"Abstract states specification and transition."
"<As for class BufferEnabledSetsActivity (enabled sets scheme, in Section 3.1)>."

always
   "Method numberOfElements is specified as an always enabled method."
   ^#(numberOfElements)
```

3.3 A Second Basic Scheme: Guards

Main alternative to enabled sets are *guards*. Intuitively, a method invocation will be blocked until the guard (boolean activation condition) associated to the method evaluates to true. Class `GuardsActivity` implements a simple and naive mechanism for guards[6]. Parameter method `nextMessage` is redefined in order to look for the first candidate message (method `isCandidateMessage:`) whose corresponding guard evaluates to true. It keeps fetching the next message and re-enqueueing it into the mailbox (method `internalReceiveMessage:`) until it finds a candidate. Last, we represent a guard associated to a method as another method (whose name is prefixed by symbol `guardOF`). Complete implementation is as follows:

```
Activity subclass: #GuardsActivity
   instanceVariableNames: ''

"Return the first candidate message. Re-enqueue it (internalReceiveMessage:) otherwise."
nextMessage
   | message |
   [message := super nextMessage.
    self isCandidateMessage: message] whileFalse:
      [address internalReceiveMessage: message].
   ^message

"A message is candidate for acceptance if its associated guard evaluates to true."
isCandidateMessage: aMessage
   ^self evaluateGuardForMessage: aMessage
```

[6] Note that we also provide refinements of this naive initial implementation scheme which do not resend messages. They use indexing messages within the mailbox and furthermore implement some optimized but safe reevaluation semantics for guards. They are implemented in subclasses of class `GuardsActivity` but won't be described in this paper.

```
"Evaluate the associated condition/guard method with the current arguments."
evaluateGuardForMessage: aMessage
   ^self perform: (self guardOfSelector: aMessage selector)
         withArguments: aMessage arguments

"Return the selector of the associated guard: the current selector prefixed by guardOF."
guardOfSelector: selector
   ^('guardOF' , selector) asSymbol
```

Example: Synchronization of a Bounded Buffer (2)

Services availability constraints on the bounded buffer are also expressed easily as guards referring to the state of the buffer:

```
GuardsActivity subclass: #BufferGuardsActivity
   instanceVariableNames: ''

"Guards: boolean activation conditions associated to methods."
guardOFget
   ^bself isEmpty not

guardOFput: item
   ^bself isFull not
```

Note that guards have the advantage over enabled sets that they may decide on the acceptance of a particular message based on some of its parameters (this is impossible with enabled sets which specify only abstractions/patterns of invocations). Also guard notations may describe intra-object synchronization, with the addition of finer grained observation of invocations status (see in Section 3.4). Furthermore, their modularity and finer grained specifications usually make them less prone to inheritance anomalies than enabled sets. However, guards have the following two main weaknesses. Their fine grain level of specification may complicate descriptions, and moreover an efficient implementation is difficult to achieve and therefore needs extra program analysis.

3.4 Second Refinement/Combination: Guards Extended with Synchronization Counters

With the addition of some fine grain mechanism observing status of current invocations, guard notations may easily express intra-object synchronization, that is control over multiple method invocations within a single active object. *Synchronization counters* [RV77] are often chosen as such a general and expressive mechanism. Main counters record the number of invocations *received*, *accepted*, and *completed* for a given method selector. Thus, synchronization counters are a direct consequence of Actalk generic events.

Class `ConcurrentActivity` provides intra-object concurrency. Its subclass `SynchroConcurrentActivity` redefines generic event methods in order to ensure their atomicity (mutual exclusion), by introducing some mutual exclusion semaphore (instance variable `mutexSemaphore`).

Consequently, the class implementing the guards/synchronization counters scheme, named `CountersActivity`, is designed as an extension/subclass of both class `SynchroConcurrentActivity` (in order to inherit concurrency and atomic events) *and* class `GuardsActivity` (to inherit guards). The only difference with class `GuardsActivity` is the redefinition of method `isCandidateMessage:` in order to ensure atomicity of a successful guard evaluation with the acceptance event (method `eventAccept:`). Class `CountersActivity` also provides dictionaries to record and consult synchronization counters data.

```
SynchroConcurrentActivity subclass: #CountersActivity
   instanceVariableNames: 'receivedDictionary acceptedDictionary completedDictionary '

initialize
   super initialize.
   "Create three dictionaries/families of counters, each indexed by the selector."
   self makeSynchroCounterDictionariesOnSelectors: bself class allScriptSelectors
```

```
"A message is candidate to be accepted if its associated guard evaluates to true."
isCandidateMessage: aMessage
   "Note that a successful guard evaluation AND the acceptance event must be atomic."
   ^mutexSemaphore critical:
       [(self evaluateGuardForMessage: aMessage)
           ifTrue: [self eventAccept: aMessage.
                    true]
           ifFalse: [false]]

"The reception event method increments the associated synchronization counter."
"(Same for methods eventAccept: and eventComplete:.)"
eventReceive: aMessage
   super eventReceive: aMessage.
   receivedDictionary at: aMessage selector
      put: (receivedDictionary at: aMessage selector) + 1

"Simulation of other useful synchronization counters."
current: selector
   "Number of current (accepted but not completed yet) invocations of message selector."
   ^(self accepted: selector) - (self completed: selector)

pending: selector
   "Number of pending (received but not accepted yet) invocations of message selector."
   ^(self received: selector) - (self accepted: selector)
```

Example: Synchronization of a Concurrent Bounded Buffer

Suppose that we now free internal concurrency of the bounded buffer[7]. We then must ensure its internal consistency, by forbidding concurrent processing of several `put:` invocations (and as well for `get` invocations). On the other hand, simultaneous processing of one `put:` *and* one `get` is allowed, as they access distinct memory sectors. Note that the number of items of the buffer is computed as the difference between completed `put:` and completed `get`, thus relying on synchronization data.

```
CountersActivity subclass: #BufferCountersActivity
   instanceVariableNames: ''

"Guards."
guardOFget
   "Only one get at once AND the buffer is not empty."
   ^(self current: #get) = 0
       and: [(self completed: #put:) - (self completed: #get) > 0]

guardOFput: item
   "Only one put: at once AND the buffer is not full."
    ^(self current: #put:) = 0
       and: [(self completed: #put:) - (self completed: #get) < bself maxSize]
```

3.5 Second Combination: Enabled Sets with Guards/Synchronization Counters

Note that enabled sets (Section 3.1) are specific to services availability constraints, whereas guards with synchronization counters (Section 3.4) add intra-object synchronization constraints. It is therefore natural to try to combine them into a single scheme, for a clear separation of services availability and intra-object concurrency. Such a mixed scheme has initially been introduced in the Dooji language by Laurent Thomas [Tho94].

Its specification is easily achieved in Actalk by combining class `EnabledSetsActivity` with class `CountersActivity` (the actual superclass) into subclass `EnabledSetsCountersActivity`. The key aspect is the *atomic* combination of the two (enabled sets *and* guards) synchronization

[7] Please keep in mind that the example of the bounded buffer is pedagogical and simple, but should *not* be considered as some significant example/granularity of object on which to introduce intra-object concurrency.

conditions (in method `evaluateGuardForMessage:`). We also combine "by hand" initialization (`initialize`) and completion event (`eventComplete:`) parameter methods, as shown below.

```
CountersActivity subclass: #EnabledSetsCountersActivity
   instanceVariableNames: 'enabledSelectors '

evaluateGuardForMessage: aMessage
   "Check both conditions (atomicity is ensured by the call by method isCandidateMessage:)."
   ^(enabledSelectors includes: aMessage selector)
      and: [super evaluateGuardForMessage: aMessage]

"By-hand combinations of methods from the two superclasses."
initialize
   super initialize.
   enabledSelectors := self perform: self initialAbstractState

eventComplete: aMessage
   super eventComplete: aMessage.
   enabledSelectors := self perform: (self nextAbstractStateAfter: aMessage selector)
```

Example: Synchronization of a Concurrent Bounded Buffer (2)

```
EnabledSetsCountersActivity subclass: #BufferEnabledSetsCountersActivity
   instanceVariableNames: ''

"Abstract states for services availability constraints."
"<As for class BufferEnabledSetsActivity (enabled sets scheme, in Section 3.1)>."

"Guards (only) for intra-object concurrency synchronization."
guardOFget
   "Only one get at once."
   ^(self current: #get) = 0

guardOFput: item
   "Only one put: at once."
    ^(self current: #put:) = 0
```

Further Method and Concurrency

With this mixed scheme, the clear separation of services availability and intra-object concurrency makes specifications more modular and consequently more reusable. Note that meanwhile we may take benefit of this separation of specifications in order to make them cooperate. As an example (taken from [Tho94]), suppose a subclass defines a method `getLast` which is like method `get` but extracts the *last* element of the bounded buffer. One optimization (increased concurrency) may be achieved by observing that one `get` *and* one `getLast` may execute concurrently (as they access distinct memory sectors) if there is *more* than one element. (Otherwise preference is given to a `get` request.)

Note that the implementation described below makes use of a technique for dynamic subpartition of the abstract state `partial` (to disable `getLast` in case of a single item). Therefore we do not need to introduce a new abstract state (thus avoiding the "partitioning of acceptable states" problem, as described in [MY93]).

```
BufferEnabledSetsCountersActivity subclass: #GetLastBufferEnabledSetsCountersActivity
   instanceVariableNames: ''

"Abstract states."
full
   ^(super full) + #(getLast)

partial
   ^bself isOne
       ifTrue: [super partial - #(getLast)]
       ifFalse: [super partial]
```

```
"Guards."
guardOFput: item
   "put: and getLast are mutually exclusive as accessing the same memory sector."
   ^(super guardOFput: item)
       and: [(self current: #getLast) = 0]

guardOFgetLast
   "As for put:. The parameter (nil) is not used/significant."
   ^self guardOFput: nil
```

3.6 Third Refinement: Guards/Synchronization Counters Extended with Generic Invocations

The previous synchronization scheme, although expressive, still cannot directly express constraints on method invocations such as "the first to come is the first to be served". Ciaran McHale et al. recently proposed some modular and expressive synchronization scheme, named *Synchronization Variables* [MBW+94], which addresses such issues. We can quickly model and implement it by augmenting previous model with the three following points.

First, we take benefit of the generic Actalk component for method invocation (class `Invocation`) to include/attach specific information such as message arrival time. We also may attach arbitrary information, e.g. job priority in order to specify priority-based specific algorithms (see below).

Secondly, we provide various iteration and predicate methods over the mailbox (that is the ordered set of pending invocations) in order to examine and compare them with current invocations (see below the application for implementation of various ordering policies).

Thirdly, we provide an optimized implementation of reevaluation semantics for guards, as opposed to the initial naive version (method **nextMessage** re-enqueueing messages) described in Section 3.3.

This enhanced activity class, named **GenericInvocationsCountersActivity**, is defined as a subclass of class **CountersActivity**. It adds instance variables: **currentInvocation** (dynamically bound to the invocation currently being checked, i.e., whose guard is being evaluated) and **reevaluationSemaphore** (to control reevaluation of guards). This class also redefines the main activity loop (parameter method **body**) to control guard evaluation and to bind instance variable **currentInvocation**. The reception event method (**eventReceive:**) is redefined in order to time stamp the invocation and (as for the two other event methods, whose redefinition is not shown here) to signal reevaluation of guards.

```
CountersActivity subclass: #GenericInvocationsCountersActivity
   instanceVariableNames: 'currentInvocation reevaluationSemaphore '

eventReceive: anInvocation
   "Assign a new time stamp to the invocation."
   super eventReceive: anInvocation.
   anInvocation arrivalTime: address nextTimeStamp.
   reevaluationSemaphore signal
```

The two basic iteration and predicate methods over pending invocations are:

```
"For all pending invocations, evaluate aBlock (whose parameter is pending invocation)."
forAllPendingDo: aBlock
   ^self mailBox do: aBlock

"Check if there is no pending invocation satisfying condition aBlock."
noPendingWith: aBlock
   ^(self mailBox detect: aBlock ifNone: [nil]) isNil
```

Example: Synchronization of a FCFS Concurrent Bounded Buffer

As example, see below the upgrading of the bounded buffer example to ensure some *first come first served (FCFS)* policy *locally* onto the **get** method:

```
guardOFget
   "Ensures FCFS policy as a further constraint (that is no prior pending get)."
   ^super guardOFget
       and: [self noPending: #get priorTo: currentInvocation arrivalTime]
```

Example: Starvation Avoidance on a Shortest Job First Served Policy

Suppose now that we alternatively constrain the bounded buffer example with some *shortest job first served* policy onto the `put:` method (this is not shown here), we may avoid starvation by dynamically decreasing the job size of a `put:` invocation each time it has been skipped:

```
eventAccept: anInvocation
   "When accepting a put:, decrease job size of all prior pending put: invocations."
   super eventAccept: anInvocation.
   anInvocation ifSelector: #put: do:
      [self forAllPending: #put: do: [:invocation |
          invocation arrivalTime < anInvocation arrivalTime
             ifTrue: [invocation decrJobSize]]]
```

Note that a more complete version of the Synchronization Variables scheme (subclass `FullGenericInvocationsCountersActivity`) provides full *recording* of current and completed invocations (as opposed to just *counting* them) in order to express more complex synchronization algorithms. Again, thanks to inheritance, we may select the level of expressiveness (and computing load) needed.

3.7 Further Extensions

We can go further, for instance, by combining the main previously described functionalities, into some new activity class (e.g., named `EnabledSetsGenericInvocationsCountersActivity`). Such a *novel* scheme is highly sophisticated and expressive, but at the cost of extra complexity and reduced efficiency. Our point in this paper is not to discuss its characteristics but in showing how our methodology/platform helps at reusing and customizing various levels of synchronization schemes, in order to produce (and implement) such refinements and variants. (You may again look at Figure 2 which summarizes the hierarchy of synchronization schemes/classes that have been developed in this section.)

4 Discussion

4.1 Evaluation

These implementations of various synchronization schemes are all integrated within a parameterized architecture and taxonomy. The benefits are in the possibilities to reuse and refine such schemes and to actually compare, apply and test them on real programs developed from the standard Smalltalk-80 environment. It is also possible to select and apply various synchronization schemes (and generally speaking various OOCP models) *locally* to various parts of a whole program/system.

Meanwhile, this experience in developing a hierarchy raises the general methodological issue of how to best extend such a taxonomy of classes while maintaining it highly modular. Currently this is the sole responsibility of the designer and implementor. As we have now some significant number of classes within the three main components taxonomies, it will be interesting to see how semi-automatic hierarchy management/reclassification mechanisms (such as [Cas92]) may help us to manage the evolution and refinement of the taxonomies.

One of current limitations of our platform is that the analysis/optimization/efficiency issues are not deeply addressed. Our pragmatic approach, to prototype and build-up up on existing software, is located somewhere between pure specifications and specific optimization concerns. One of our future works is in expanding the platform tools to better account for the efficiency concern.

4.2 Related and Future Work

Our survey of various synchronization schemes has some similarities with the analysis and proposal of Satoshi Matusoka et al. [MY93]. They discuss and analyze various synchronization schemes and also use a multi-paradigm approach for their proposal to solve the "inheritance anomaly" phenomenon. Their reference work has been an influence to the work described here. Meanwhile, our work differs from theirs in the two following ways: we consider the issue of intra-object concurrency, and we propose a software workbench and programming model libraries to help the designer/programmer at (himself) designing and implementing customized prototype synchronization schemes.

The pros of building object-oriented (and more specifically Smalltalk-based) generic platforms for classifying various programming constructs has also been demonstrated by platforms such as Simtalk (for modeling various simulation schemes [Béz87]) and others (Classtalk, Prototalk…). A generic scheduler has also been developed by Loïc Lescaudron [Les92], as part of the Actalk project, to classify and parameterize various scheduling policies.

The component-based architecture of an Actalk active object is close to the component-based meta-architecture of CodA, designed by Jeff McAffer [McA95]. CodA provides finer grain components and more refined interface between them at the cost of more complexity. We definitely intend to study the porting of our synchronization libraries into CodA in order to cross-fertilize both projects.

There are other alternatives to components and methods decomposition, as for instance with parameterizing the invocation of some methods via arguments, as advocated by the Hermes/ST architecture [FHR93]. This is also related to the issue of having a minimal kernel and extending it with more complex protocols and mechanisms versus having a bigger kernel offering more protocols not necessarily all used at the kernel level. In the case of Actalk, the kernel has been designed a while ago to be simple and efficient, as its initial motivation was mainly didactic. The scope of Actalk has also been restricted as it does not address distribution aspects (as for CodA) or fault-tolerance aspects (as for Hermes/ST) but concentrates onto language design.

Alternatives to represent various OOCP designs are some more formal approaches, as the object calculus proposed by Oscar Nierstrasz [Nier93]. Note that our more pragmatic approach allows us experiments on actual programs within a sophisticated programming environment (Smalltalk-80 based) to help developing and monitoring them.

Conclusion

In this paper we have described the pros of using a platform for studying and designing various synchronization schemes for object-oriented concurrent programs. We first introduced the key architectural aspects of the Actalk architecture/platform. We then implemented successive refinements and combinations of synchronization schemes in order to show the expressive power of our platform. We finally evaluated our experience and related it to other results.

We now prospect at using our platform and experience not only for comparing various design alternatives in a local way, but also to study the global effects on the whole architecture of a software system. (At present, we are notably interested in the experience/study of concurrent/distributed multi-layered knowledge-based systems).

Acknowledgements

We would like to thank Jeff McAffer for many general discussions on OOCP and reflection subjects and for his helpful comments on the presentation of this paper.

References

[AS83] G.R. Andrews and F.B. Schneider, Concepts and Notations for Concurrent Programming, *ACM Computing Surveys*, Vol. 15, No 1, pages 3–43, March 1983.

[Béz87] J. Bézivin, Some Experiments in Object-Oriented Simulation, in *Proc. of OOPSLA'87*, Special Issue of ACM SIGPLAN Notices, Vol. 22, No 12, pages 394–405, December 1987.

[BH94] S. Billet-Coat and D. Herin-Aimé, A Multi-Agent Architecture for an Evolving Expert System Module, in *5th International Conference on Database and Expert Systems Applications (DEXA'94)*, ed. by D. Karagiannis, LNCS, No 856, Springer-Verlag, pages 581–590, September 1994.

[BFS91] T. Bouron, J. Ferber and F. Samuel, MAGES: a Multi-Agent Testbed for Heterogeneous Agents, in *Decentralized Artificial Intelligence*, Vol. II, ed. by Y. Demazeau and J.-P. Muller, North-Holland, 1991.

[Bri89] J.-P. Briot, Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. in *Proc. of ECOOP'89*, ed. by S. Cook, Cambridge University Press, pages 109–129, July 1989.

[Bri94] J.-P. Briot, Modélisation et Classification de Langages de Programmation Concurrente à Objets : l'Expérience Actalk, *LITP Research Report 94/59*, LITP, Institut Blaise Pascal, France, October 1994.

[Cas92] E. Casais, An Incremental Class Reorganization Approach, in *Proc. of ECOOP'92*, ed. by O. Lehrmann Madsen, LNCS, No 615, Springer Verlag, pages 133–152, June 1992.

[FHR93] M. Fazzolare, B.G. Humm, and R.D. Ranson, Object-Oriented Extendibility in Hermes/ST, a Transactional Distributed Programming Environment, in *Proc. of the ECOOP'93 Workshop on Object-Based Distributed Programming*, ed. by R. Guerraoui, O. Nierstrasz, and M. Riveill, LNCS, No 791, Springer-Verlag, pages 241–261, 1994.

[KG84] M. Kang and D.D. Grant, A Kernel Mechanism for Simulating an Object-Oriented Process Sensitive Software Engineering Environment, in *TOOLS Pacific'94 (TOOLS 15)*, pages 57–67, TOOLS/ISE, Fall 1994.

[LBB91] L. Lescaudron, J.-P. Briot and M. Bouabsa, Prototyping Programming Environments for Object-Oriented Concurrent Languages: a Smalltalk-Based Experience, in *Proc. of TOOLS USA'91*, Interactive Software Engineering - Prentice-Hall, page 449–462, August 1991.

[Les92] L. Lescaudron, Prototypage d'Environnements de Programmation pour les Langages à Objets Concurrents : une Réalisation en Smalltalk-80 pour Actalk, *Thèse d'Université*, LITP, University Paris VI - CNRS, *LITP Thesis Report TH93.11*, LITP, Institut Blaise Pascal, France, May 1992.

[MY93] S. Matsuoka and A. Yonezawa, Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, Chapter 4 of *Research Directions in Concurrent Object-Oriented Programming*, ed. by G. Agha, P. Wegner and A. Yonezawa, Mit Press, pages 107–150, 1993.

[McA95] J. McAffer, Meta-level Programming with CodA, to appear in *ECOOP'95*, ed. by W. Olthoff, LNCS, Springer-Verlag, July 1995.

[MBW+94] C. McHale, S. Baker, B. Walsh, and A. Donnelly, Synchronization Variables, *Technical Report*, TCD-CS-94-01, Dept. of Computer Science, University of Dublin, January 1994.

[Nier93] O. Nierstrasz, Composing Active Objects, Chapter 5 of *Research Directions in Concurrent Object-Oriented Programming*, ed. by G. Agha, P. Wegner and A. Yonezawa, Mit Press, pages 151–171, 1993.

[OOCP87] Object-Oriented Concurrent Programming, ed. by A. Yonezawa and M. Tokoro, *Computer Systems Series*, MIT Press, 1987.

[RV77] P. Robert and J.-P. Verjus, Towards Autonomous Descriptions of Synchronization Modules, in *Information processing 77: Proc. of IFIP'77*, ed. by B. Gilchrist, North Holland, pages 981–986, August 1977.

[Tho94] L. Thomas, Inheritance Anomaly in True Concurrent Object Oriented Languages: A Proposal, in *Proc. of IEEE TENCON'94*, pages 541–545, August 1994.

Contents