Actalk: A Framework for Object-Oriented Concurrent Programming - Design and Experience

Jean-Pierre Briot
Laboratoire d'Informatique de Paris 6 (LIP6)
Université Paris 6 - CNRS
Case 169, 4 place Jussieu, 75252 Paris Cedex 05, France
E-mail: `Jean-Pierre.Briot@lip6.fr`

ABSTRACT. This paper describes a framework for modeling, classifying, *and* specializing *various object-oriented concurrent programming (OOCP) languages constructs and schemes. This framework, named* Actalk, *is implemented and integrated within the Smalltalk-80 programming environment. The architecture of the framework is based on the notions of* components, parameterization *and* inheritance. *The* kernel *of the architecture is decomposed into a set of kernel* component classes *and associated* parameter methods, *to be redefined in various subclasses. Various* extensions, *defined as* subclasses *of the kernel classes, simulate various OOCP languages and constructs. They implement various models of* activity *(serialized, concurrent, with implicit or explicit message acceptance...),* communication *(synchronous, eager reply, time stamped...), and* synchronization schemes *(guards, abstract states transitions, synchronization counters...). At first, our framework helps in classifying and experimenting with various synchronization and communication models for a given program, by changing and specializing various models/components. Secondly, it helps in reusing them and building-up on previous expertise in order to derivate enhanced or novel constructs or schemes. To illustrate the possibilities in this paper, we progressively develop various synchronization schemes by successive* refinements *and* combinations, *starting from two foundational synchronization schemes (namely* enabled sets *and* guards*).*

Keywords: concurrency, programming, languages, communication, synchronization, schemes, framework, platform, components, parameterization, refinement, combination, implementation, Smalltalk-80.

1 Introduction

Object-oriented concurrent programming (OOCP in short [YT 87]) is an important methodology for development of parallel, distributed and open applications [BGY 96]. It is a generalization of object-oriented programming taking into account multiple activities. OOCP provides both: (1) a methodology for the *decomposition* of complex programs which may expressed more naturally as a coordination of multiple interacting entities, and (2) a means for exploiting perspectives of speed-up as offered by new parallel and distributed computing architectures.

As a consequence of the success of this approach, various programming models, constructs and mechanisms have been proposed (e.g., see in [YT 87], [AWY 93], [BGY 96]). They reflect a wide variation of concerns and domains. This variety of models, and the difficulty to compare them by abstracting from various associated terminologies, syntax and implementational foundations, led us to design some small comprehensive classification testbed in 1988 [Bri 89]. In a second step, this testbed was later expanded in 1994 into a full framework [Bri 96]. After several years of successive developments, experiments, usage by various communities for different application domains and concerns, we would like to summarize the results and lessons learned. This paper aims at providing such description and assessments.

1.1 Outline

This paper is organized as follows. In Sect. 2, we first discuss the motivations for the Actalk platform. In Sect. 3, we then discuss the precise objectives of the platform and the main design decisions we made. Section 4 describes the architecture and more specifically the way the kernel is decomposed and parameterized. In Sect. 5, we progressively develop various schemes by successive *refinements* and *combinations*, starting from two foundational synchronization schemes (namely *enabled sets* and *guards*). Section 6 quickly summarizes and evaluates our approach and results, and then compares them to other relevant works before concluding this paper.

2 Context and Motivations

2.1 Object-Oriented Concurrent Programming

Object-oriented (concurrent) programming (OOCP) 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.

OOCP *generalizes* standard (sequential) object-oriented programming. The basic idea is to integrate object notions with the main abstractions of concurrent programming.[1] It identifies the notion of object with the notion of activity,

[1] The objective of this integrative approach is to provide a simple and unified conceptual model to the programmer. Note that this is not the only way to relate object concepts with concurrency, parallelism, and distribution concerns. [BGL 98] considers three main approaches (applicative, integrative, reflective) and studies their specificities, differences as well as their complementary aspects and levels.

leading to the notion of an *active object*. It also identifies message passing between objects with the synchronization of the sender (client) onto the receiver (server). This eases the issue of synchronization of concurrent programs, as synchronization of requests becomes transparent to the client.

2.2 Various Dimensions

Based on these general concepts, various alternative programming models, constructs and mechanisms have been proposed in various programming languages and systems [YT 87, AWY 93, BGY 96]). These various systems reflect a wide variation of concerns and domains. Possible variations on programming language design may be grouped into various aspects, or dimensions, such as:

- *communication*: is communication between objects synchronous or asynchronous ?, with a single reply or multiple replies ? are there priority schemes ?, etc.
- *activity*: is acceptance of messages implicit or explicit ?, is there intra-object concurrency (i.e., may an object compute several requests simultaneously) ?, etc.
- *synchronization/coordination*: how does an active object control message selection and activation ? depending on what information ? (its state, current invocations, required services, etc.) and along which scheme ?[2]

2.3 Motivations

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 testbed.

This testbed has been developed in, and integrated within, the Smalltalk-80 programming environment. It is named *Actalk*, which stands for *act*ive objects (or *act*ors) in Small*talk*. It is a framework and is based on the notions of *components*, *parameterization*, and *inheritance*. Various component classes correspond roughly to various dimensions, as defined above.

Motivations are both:

- to help at the *analysis* and *classification* of different aspects of various formalisms (computation models, programming langagues, constructs, schemes), of which different syntax and implementation frameworks may hide the semantics.

[2]Finding schemes which allow easily reusable specifications is actually non trivial. This problem has been named the *"inheritance anomaly phenomenon"* by Satoshi Matsuoka [MY 93]. Although this paper is not about proposals for solving the anomaly, we will show in Sect. 5 that our platform already includes most of schemes which have been proposed, and moreover helps at adapting them.

- to help at the *selection* and *comparison* with various computation models, by associating various components (e.g., various synchronization schemes, taken from the existing library of components) to a given program.
- to ease at the *design* of new formalisms, by *refinement* and *combination* of existing components. (An example of successive refinements and combinations will be developed in Sect. 5).
- to provide an environment for active experiment, with a rich *programming environment* including visualization and debugging tools.

3 Objectives and Design Decisions

3.1 Objectives

To design a system for integrating various OOCP models and languages in a single environment, we had the following objectives in mind:

- *uniformity*. Various object-oriented concurrent programming computation models, languages constructs, and schemes are grouped and classified into one *single* framework.
- *minimality*. Actalk is based on a *kernel* which provides the basic and default semantics of OOCP languages.
- *modularity*. The kernel is decomposed into a set of kernel *component classes* and their associated *parameter methods*.
- *extensibility and expressivity*. Various extensions, defined as *subclasses* of the kernel classes and which specialize some of the parameter methods, simulate various OOCP languages and constructs.
- *simplicity*. We are not concerned in this platform with considerations for optimization or formal specifications, as they may complexify the implementation.
- *integration*. Actalk is integrated within the Smalltalk-80 programming environment. Thus the user may combine various object-oriented concurrent programming styles with standard object-oriented programming, as provided by Smalltalk-80, to reuse standard programs and to express various program granularities.
- *experiment*. Actalk includes a library of various component classes describing various language constructs, models of activity, synchronization, and communication, and associated example programs. Standard Smalltalk-80 programming environment provides various development tools to develop schemes and programs.[3]

[3]In fact some of the Smalltalk-80 standard tools, such as the MVC user interface framework, the scheduler, and the debugger, need some adjustements for dealing properly with

3.2 Choosing Smalltalk

Smalltalk-80 was chosen as a foundation for its high-level, flexible and rich environment. Smalltalk-80 provides all entities needed to build active objects (objects, classes), and to manage their activities (processes, message perform), their communication (messages, queues), and their synchronization (semaphores, shared queues) [BG 96].[4]

4 Architecture of the Actalk Framework

4.1 General Architecture

The architecture of Actalk includes a *kernel* which models basic OOCP semantics (that is, serialized active 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 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 Sect. 6.3, when comparing our platform to a few other similar systems.

4.2 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 standard 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 `ActorObject` implements the Actor model of computation construct `replace` for specifying the replacement behavior of an actor [AGH 86]. Class `Abcl1Object` implements the ABCL/1 language

active objects. A past project has developed prototype extensions of Smalltalk-80 standard tools, such as an extended MVC model for active objects, and a generic scheduler and its associated visualization tools [LES 92].

[4] One partial meta-representation of Actalk, where Actalk active objects may be described by a set of active objects, has been developed by Sylvain Giroux and named ReActalk [GS 91]. ReActalk increases the flexibility of Actalk through reflection, at the cost of a little more complexity and little less efficiency.

construct `wait-for`, to explicit wait for some message pattern [YT 87, pages 55–89].)

- Class `Activity` describes the *internal activity* of the active object. This class provides autonomy[5] 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 to which messages will be sent. This class defines the way message transmissions will be interpreted. Its subclasses may implement various types of communication.[6]

This decomposition allows the independent modeling of various dimensions of active objects. One can decouple the actual program from a given communication model (e.g., with eager reply), and from a given model of activity (e.g., with intra-object concurrency) and synchronization scheme[7] (e.g., guards). Meanwhile, combination of arbitrary components may lead to inconsistencies. Therefore the Actalk platform includes some simple compatibility specification and verification mechanism to keep some safeguards [Bri 94].

Note that the kernel actually provides two more kernel component classes: class `MailBox` which represents the message queue, and class `Invocation` (a subclass of standard class `Message`) which represents a method invocation. 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 for invocation management, e.g., with time stamps (to be described in Sect. 5.5).

4.3 Relations between Components

To define a class of active objects, the programmer should define this class as inheriting from class `ActiveObject` (and not `Object` as usual). 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

[5] through a (light weight) standard Smalltalk-80 process.

[6] In Actalk, the receiver defines the semantics of message passing (synchronous, asynchronous...). This simple scheme provides modularity and encapsulation, as well as simple and efficient implementation. It may be extended into a more general framework where the type of communication is negotiated between sender and receiver, at the cost of more complexity and less efficiency.

[7] 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 [McH 94]. This also eases comparison of various synchronization schemes on actual examples by just changing the synchronization class while keeping the same behavior/program.

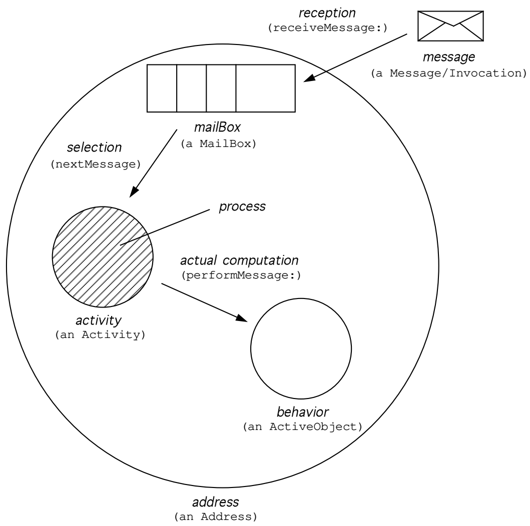

Figure 1: Components of an Actalk active object

methods (e.g., parameter method `addressClass` specifying the associated address component class, as later detailed in Table 1). This follows the *Factory method* design pattern [GHJ+ 95].

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

4.4 The Activity Kernel Component Class and its Parameterization

Class `Activity` defines two instance variables: `address`, to reference the associated address component, and `bself` (as for *b*ehavior *self*), to reference the associated behavior component, of the active object. Parameter methods of class `Activity` are summarized in Table 1.

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

method selector	*parameter*	*default value*	*example of redefinition*
`initialize`	*initialization*	none	initialize synchronization counters (see Sect. 5.3)
`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 model of 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 Sect. 5.1)
`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`)
`invocation-ClassFor:`	*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 Sect. 5.1)

4.5 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: *reception/acceptance/completion* of an invocation. They are respectively named: `eventReceive:/Accept:/Complete:`. (They take the current invocation as their argument.)

The user may redefine them to attach actions to a given class of active objects, e.g.: for 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., for computing post actions as with the POOL language [YT 87, pages 199–220]). Last, they are also very useful for managing intra-object synchronization. Subclass `SynchroConcurrentActivity` specializes them in order to ensure their atomicity (mutual exclusion), as we will see in Sect. 5.3.

4.6 Layered Architecture

In complement of its component-based architecture (Sect. 4.2), and its event-based architecture (Sect. 4.5), the kernel is also designed along some layered architecture.[8] The three successive layers[9] help at separating *basic properties*, *added facilities*, and *(public) entry points* for the user.

The first layer defines *minimal* versions of the three main component classes of the kernel, including basic methods and parameter methods. The second layer adds more facilities, notably the event mechanism. The third layer defines the three kernel component classes `ActiveObject`, `Activity` and `Address`, to be subclassed by the programmer. They also include some user-intended functionalities (tracing, compatibility constraints...).

4.7 Libraries

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 [AGH 86], ABCL/1 explicit wait for a message pattern [YT 87, pages 55–89], POOL concept of body and post actions construct [YT 87, 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)... ;

[8] This shows an example of combining three different software architectural styles [SG 96] within a single system.

[9] They were introduced by Loïc Lescaudron [LES 92].

- and *synchronization schemes*: the focus of this paper. Part of the corresponding taxonomy is detailed in Sect. 5. Other schemes include: explicit message acceptance (as in POOL and Eiffel//), method suspension (as in Portable ConcurrentSmalltalk), reflective framework (as in OCore)...

4.8 Other Developments and Applications

With its libraries and taxonomies of various OOCP languages characteristics, the Actalk framework provides a comprehensive view of design alternatives and mechanisms. Actalk has been used as a tool for teaching and experiments by various people, inside Paris 6 University, but also in other places, notably in the early 90's during graduate courses led by Jean Bézivin at University of Nantes where students projects developed many experiments.

The kernel of Actalk has also been used as component or foundation for several projects and domains, such as simulation of software engineering process models [KG 94], distributed constraint satisfaction algorithms (e.g., [GHÉ 94]), the study of exception mechanisms for object-oriented concurrent programming, etc. One notable field of application has been for construction by various teams of various prototype multi-agent systems, themselves applied to various domains (eco-systems simulation, distributed problem solving [BFS 91], natural language processing [NH 96], knowledge acquisition [BH 94]...). A recent example multi-agent platform, and the way it is built up from Actalk active objects, is described in [GB 98].

5 A Developed Example: Modeling Synchronization Schemes

We will now model and 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 Sect. 5.3), 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 Fig. 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.

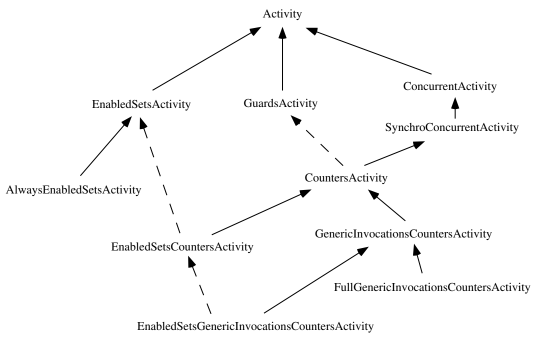

Figure 2: Hierarchy of synchronization schemes/classes (sample)

5.1 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 (also named *behavioral abstractions*) are abstractions/names to which enabled sets will be assigned. (In the canonical example of the bounded buffer, `put:` requests should be disabled while the buffer is full. This will be expressed with an abstract state `full` having only {`get`} as its corresponding enabled set.) After selecting and computing the current method invocation, a state transition function will 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:)`).[10] An abstract state is represented by a method returning an enabled set. Implementation only redefines three of class `Activity` parameter methods: `initialize`, `nextMessage`, and *completion* 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."
```

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

```
   enabledSelectors := self perform: self initialAbstractState

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

"Generic event method associated to completion of the invocation (computation)."
eventComplete: aMessage
   super eventComplete: aMessage.
   "Abstract state transition: computing 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 (1)

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 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 synchronization component of the bounded buffer is specified in class `BufferEnabledSetsActivity`, defined as a subclass of class `EnabledSetsActivity`:

```
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]]
```

Note that the implementation of the bounded buffer behavior/program (trivial and here out of scope) is not shown in this paper. We assume that the behavior implements methods `put:` and `get`. We also assume that it implements the two predicate methods `isEmpty` and `isFull`, and the accessor method `maxSize` (to consult the maximum size). As opposed to methods `put:` and `get`, these latter methods are not declared in the external interface of the active object (i.e. they won't be enabled). They are for internal use only, so that the activity/synchronization component may (namely, method `nextAbstractStateAfter:`) consult the state of the buffer behavior (referenced by instance variable `bself`, see Sect. 4.4).

5.2 A Second Basic Scheme: Guards

The 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.[11] 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. Finally, we represent a guard associated to a method as another method (whose name is prefixed by symbol `guardOF`). The *complete* implementation is as follows:

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

"Return the first candidate message, otherwise re-enqueue it''
   (method internalReceiveMessage:)."
nextMessage
   | message |
   [message := super nextMessage.
    self isCandidateMessage: message] whileFalse:
```

[11] Note that we also provide refinements of this naive initial implementation scheme without resending 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.

```
      [address internalReceiveMessage: message].
   ^message

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

"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: selector prefixed by guardOF."
guardOfSelector: selector
   ^('guardOF' , selector) asSymbol
```

Example: Synchronization of a Bounded Buffer (2)

Services availability constraints on the bounded buffer are easily expressed 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. But efficient implementations of guards is more difficult to achieve.

5.3 A First Refinement: 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* [RV 77] 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. Actually, synchronization counters are a direct consequence of (Actalk) generic events.

We also need a specialized activity class providing intra-object concurrency, as implemented by class `ConcurrentActivity`. Its subclass `SynchroConcurrentActivity` redefines generic event methods in order to ensure their atomicity (mutual exclusion), by introducing some mutual exclusion semaphore (instance variable `mutexSemaphore`).

Now, we may implement synchronization counters (class `CountersActivity`) as a 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` (Sect. 5.2) 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 implements dictionaries to record and consult synchronization counters data.

```
SynchroConcurrentActivity subclass: #CountersActivity
   instanceVariableNames: 'receivedCounterDictionary
          acceptedCounterDictionary completedCounterDictionary '

initialize
   super initialize.
   "Create a dictionary per family of counters, indexed by each selector."
   self makeSynchroCounterDictionariesOnSelectors:
          bself class allScriptSelectors

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

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

"Consultation of the reception synchronization counter."
"(Same for methods accepted: and completed:.)"
received: selector
   "Number of received invocations of message selector."
   ^receivedCounterDictionary at: selector

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

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

Example: Synchronization of a (Concurrent) Bounded Buffer (3)

Suppose that we now free the internal concurrency of a bounded buffer.[12] 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 only 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]
```

5.4 A Combination: Enabled Sets with Guards/Synchronization Counters

Note that enabled sets (Sect. 5.1) are specific to services availability constraints, whereas guards with synchronization counters (Sect. 5.3) 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 [THO 94].

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 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
```

[12] 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.

```
   "Check both conditions."
   "(Atomicity is ensured by the call from method isCandidateMessage:,"
   "   see in Sect. 5.3.)"
   ^(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 (4)

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

"Abstract states for services availability constraints."
**As for enabled sets scheme, class BufferEnabledSetsActivity in Sect. 5.1**

"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
```

With this mixed scheme, the clear separation of services availability and intra-object concurrency makes specifications more modular and consequently more reusable.

5.5 Further Extensions

The previous synchronization scheme (Sect. 5.4), although expressive, still cannot directly express constraints on method invocations such as "shortest job first served" policy. [McH 94] proposed some modular and expressive synchronization scheme, named *Synchronization Variables*, which addresses such issues. We can quickly model and implement their scheme by augmenting previous synchronization counters model (1) by attaching specific information such as message arrival time or job priority onto invocations and (2) by providing various iteration and predicate methods over the mailbox (that is the ordered set of pending invocations). Such examples of refinements, which can express fine grain policies while ensuring liveness, have been implemented and described in [Bri 96]

Note that our point in this paper is not to discuss the characteristics of synchronization schemes with growing complexity but in showing how our framework 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 Fig. 2 which summarizes the hierarchy of synchronization schemes/classes that have been partially developed in this section.)

6 Evaluation, Related and Future Work

6.1 Evaluation of Current Framework

As we shown, implementations of various languages models, constructs and synchronization schemes have been rather easily integrated within a single framework. The benefits are in the possibilities to reuse and refine such models, constructs and 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* and *computational contexts* 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.[13] 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 [Cas 92]) may help us to manage the evolution and refinement of the taxonomies.

6.2 Evaluation of Evolution

Note that the initial version of Actalk in 1988 was much smaller than current framework. The 88 kernel included only two component classes (activity and behavior were not separated yet) and much less methods. In 1994 we developed a large number of class extensions of the kernel to implement numerous OOCP computation models, languages and synchronization schemes. Current Actalk framework emerged through the identification of the stable parts (kernel components and their basic methods) and their parametrization (parameter methods). This complexified the architecture albeit increased the possibilities. By comparing initial platform and current framework, we note that the initial platform got the larger use by numerous teams who used the kernel as a basis for implementing systems based on active objects. Thus, by using terms defined by Christopher Alexander [Ale 75] and highlighted by Richard Gabriel

[13] We therefore use hierarchic parameterization of the architecture, i.e. decomposing further parameter methods as needed within the hierarchy, as discussed in [Bri 94].

[Gab 96], we could say that although the 94 framework provided more extensive and systematic modeling abilities, *habitability* and *piecemeal growth* was better achieved for the first - lighter - version of Actalk.

6.3 Related Work

The pros of building object-oriented (and more specifically Smalltalk-based) generic platforms for classifying various programming constructs has also been demonstrated by other platforms, such as Simtalk (for modeling various simulation schemes [Béz 87]) as well as others (Classtalk, Prototalk...). A generic scheduler has also been developed, as part of the Actalk project, by Loïc Lescaudron to classify and parameterize various scheduling policies [LBB 91, Bri 93].

Alternatives to represent various OOCP designs are some more formal approaches, as the object calculus proposed by Oscar Nierstrasz [Nie 93]. Note that our pragmatic approach allows experiments with actual programs within a sophisticated programming environment (Smalltalk-80 based) to help at developing and monitoring them [LBB 91].

The component-based architecture of an Actalk active object is close to the component-based meta-architecture of CodA, designed by Jeff McAffer [McA 95]. CodA provides finer grain components and more refined interface between them, but at the cost of more complexity.

There are also 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 [FHR 93]. 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 (see also previous discussion in Sect. 6.2). 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 [Bri 89]. 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 on OOCP language design.

6.4 Future Work

Besides the areas and directions for future work already mentioned, an important and general issue is the combination of various aspects/descriptions of a computational behavior. The decomposition of the Actalk architecture in orthogonal components, and their further decomposition in parameter methods, help at such combination. Meanwhile, it reaches some limitations when combining differents versions of a same component (in the context of this paper, the activity/synchronization component). The programmer may have then to rely on some amount of explicit combination. A finer grain decomposition of components, as proposed by CodA [McA 95], brings more independence and modularity, but it also still relies on a single component to cover activity and synchronization concerns. Our belief is that it is difficult anyway to further decompose a complex aspect, such as activity/synchronization, in fully orthogonal pieces. (In other words, we ultimately reach some atoms or even quarks.) Therefore, we believe that we cannot avoid the general problem of composing non fully orthogonal components, and that we should develop some rationale and methodology for doing so. Some starting propositions may for instance be found in [McA 95].

7 Conclusion

In this paper we have described the design of a framework for object-oriented concurrent programming, its architecture, and examples of use. We implemented successive refinements and combinations of synchronization schemes in order to show the expressive power of our platform. We finally evaluated our experience, related it to other works, and pointed future areas of investigation.

Latest version of Actalk (including some documentation) is available through anonymous ftp/http: "`ftp://ftp.lip6.fr/lip6/softs/actalk/actalk.html`" or: "`ftp://ftp.yl.is.s.u-tokyo.ac.jp/pub/actalk/actalk.html`".

References

[Agh 86] G. Agha, *Actors: a Model of Concurrent Computation in Distributed Systems*, *Series in Artificial Intelligence*, MIT Press, 1986.

[AWY 93] G. Agha, P. Wegner, and A. Yonezawa (editors), *Research Directions in Concurrent Object-Oriented Programming*, Mit Press, 1993.

[Ale 75] C. Alexander, "The Oregon Experiment," Oxford University Press, 1975.

[AS 83] G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, Vol. 15, n^o 1, March 1983, pages 3–43.

[Béz 87] J. Bézivin, "Some Experiments in Object-Oriented Simulation," *Proc. of OOPSLA'87*, Special Issue of ACM SIGPLAN Notices, Vol. 22, n^o 12, December 1987, pages 394–405.

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

[BFS 91] T. Bouron, J. Ferber, and F. Samuel, "MAGES: a Multi-Agent Testbed for Heterogeneous Agents," *Decentralized Artificial Intelligence*, Vol. II, edited by Y. Demazeau and J.-P. Muller, North Holland, 1991.

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

[Bri 93] J.-P. Briot, "Object-oriented design of a generic scheduler," *Object-Oriented Computing II* (Proc. of JSSST WOOC'93), edited by A. Yonezawa, S. Matsuoka, and W. Kato, Lecture Notes/Software Science, n^o 6, Kindai-Kagaku-Sha, Japan, 1994, pages 73–81.

[Bri 94] J.-P. Briot, "Modélisation et Classification de Langages de Programmation Concurrente à Objets : l'Expérience Actalk," *LITP Research Report*, n^o 94/59, Institut Blaise Pascal, France, October 1994. (Also in *Proc. of Conference "Langages et Modèles à Objets (LMO'94),"* INRIA/IMAG/PRC-IA, Grenoble, France, October 1994, pages 153–165.)

[Bri 96] J.-P. Briot, "An Experiment in Classification and Specialization of Synchronization Schemes," *Object Technologies for Advanced Software* (Proc. of ISOTAS'96), edited by K. Futatsugi and S. Matsuoka, LNCS, n^o 1049, Springer-Verlag, March 1996, pages 227–249.

[BG 96] J.-P. Briot and R. Guerraoui, "On the Use of Smalltalk for Concurrent and Distributed Programming," *Informatik/Informatique*, Swiss Professional Informatics Societies, Switzerland, Special Issue on "Smalltalk (Languages, Tools, Environments)," edited by R. Guerraoui, February 1996.

[BGL 98] J.-P. Briot, R. Guerraoui, and K.-P. Löhr, "Concurrency and Distribution in Object-Oriented Programming," *ACM Computing Surveys*, Vol. 30, n^o 3, September 1998, pages 291–329.

[BGY 96] J.-P. Briot, J.-M. Geib, and A. Yonezawa (editors), *Object-Based Parallel and Distributed Computation*, LNCS, Springer-Verlag, 1996.

[Cas 92] E. Casais, "An Incremental Class Reorganization Approach," *Proc. of ECOOP'92*, edited by O. Lehrmann Madsen, LNCS, n^o 615, Springer-Verlag, June 1992, pages 133–152.

[FHR 93] M. Fazzolare, B.G. Humm, and R.D. Ranson, "Object-Oriented Extendibility in Hermes/ST, a Transactional Distributed Programming Environment," *Proc. of the ECOOP'93 Workshop on Object-Based Distributed Programming*, edited by R. Guerraoui, O. Nierstrasz, and M. Riveill, LNCS, n^o 791, Springer-Verlag, 1994, pages 241–261.

[Gab 96] R.P. Gabriel, "Patterns of Software - Tales from the Software Community," Oxford University Press, 1996.

[GHJ+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Design*, Addison-Wesley, 1995.

[Ghé 94] K. Ghédira, "A Distributed Approach to Partial Constraint Satisfaction Problems", *Proc. of the 6th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'94)*, LNCS, n^o 1069, Springer-Verlag, 1996.

[GS 91] S. Giroux and A. Senteni, "ReActalk, a Reflective Version of Actalk," *OOPSLA'91 Workshop on Reflection and Metalevel Architectures*, Xerox PARC Technical Report, October 1991.

[GB 98] Z. Guessoum and J.-P. Briot, "Building Agents as an Extension of Active Objects - Application to the Simulation of Economic Models," *Proceedings of the Conférence Africaine sur la Recherche en Informatique (CARI'98)*, INRIA, October 1998.

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

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

[Les 92] L. Lescaudron, "Prototypage d'Environnements de Programmation pour les Langages à Objets Concurrents : une Réalisation en Smalltalk-80 pour Actalk," *Thèse d'Université*, TH93.11, LITP, Université Paris VI - CNRS, France, May 1992 (in french).

[MY 93] S. Matsuoka and A. Yonezawa, "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages," *Research Directions in Concurrent Object-Oriented Programming*, edited by G. Agha, P. Wegner, and A. Yonezawa, Mit Press, 1993, pages 107–150.

[McA 95] J. McAffer, "Meta-level Programming with CodA," *Proc. of ECOOP'95*, edited by W. Olthoff, LNCS, n^o 952, Springer-Verlag, August 1995, pages 190–214.

[McH 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.

[NH 96] P. Neuhaus and U. Hahn, "Restricted Parallelism in Object-Oriented Parsing," *Proceedings of the 16th International Conference on*

Computational Linguistics (COLING'96), Vol. 1, Kyoto, Japan, August 1996, pages 379–385.

[NIE 93] O. NIERSTRASZ, "Composing Active Objects," in [AWY 93], pages 151–171.

[RV 77] P. ROBERT and J.-P. VERJUS, "Towards Autonomous Descriptions of Synchronization Modules," *Proc. of IFIP'77*, edited by B. Gilchrist, North-Holland, pages 981–986, August 1977.

[SG 96] M. SHAW and D. GARLAN, *Software Architectures: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

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

[YT 87] A. YONEZAWA and M. TOKORO (editors), *Object-Oriented Concurrent Programming, Computer Systems Series*, MIT Press, 1987.