Modeling and Classifying OOCP Languages and Constructs

Jean-Pierre BRIOT*

Dept. of Information Science
The University of Tokyo
Hongo, Bunkyo-ku, Tokyo 113, Japon

`briot@is.s.u-tokyo.ac.jp`

This is a set of slides used as a support for a talk.
Eventually will turn into a full paper.
Made available as info on current Actalk version.
May 1994.

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

Contents

1 Object-Oriented Concurrent Programming

Object-oriented concurrent programming (OOCP at short) is a promising methodology to match novel challenges of distributed and open computing.

OOCP describes computation through a collection of small self-contained modules (named *objects*) which compute and interact concurrently through some unified communication protocol (named *message passing*).

OOCP provides a good foundation for decomposing programs and running them efficiently onto multiprocessors.

Objects and Concurrency

OOCP achieves integration of Object-Oriented programming (OOP) with concurrency.

It identifies objects as the unit of activity and synchronization, and associates synchronization between objects at the message passing level.

Resulting unification of an object with an activity is named *active object*.

This unification achieves integration of concepts of OOP and concurrency and frees the programmer from having to explicitly take care of most of synchronization discipline.

This also preserves modularity and simplicity of OOP while enforcing self-containedness and autonomy of objects.

Advantages

Main advantages of OOCP may be stated as following:

- **high-levelness**
- **implicit decomposition of concurrent activities**
- **transparent synchronization**
- **locality and self-containedness**
- **dynamicity and openness**
- **multi-granularity**

Applications

Object-oriented concurrent programming is being applied to a growing number of fields. Applications are specially growing strong in the following fields: distributed operating systems, (distributed) artificial intelligence, distributed simulation, distributed data-bases, office information systems, real-time systems and (distributed) process control.

Significant results may also be found in other fields, e.g., natural language processing, and computer music.

OOCP has also main impact on new multi-processor architectures, like the J-Machine.

2 Motivations

Variety of OOCP Models and Languages

- Various OOCP models, languages, and constructs have been and are still being introduced.
- Even a single language (e.g., ABCL, POOL, ConcurrentSmalltalk...) may have successive variants.
- Various proposals represent various compromises regarding various application domains and architectures targeted.

Difficulty to Relate and Combine Them

- It is not always easy to relate and compare these various proposals.
- Various syntax and implementation platforms often obscure their relations.
- It is not trivial to reuse these various models and moreover to experiment with combining them.

Three Dimensional Variation Space

Variations are mainly along three axes:

- **communication**
  what are possible message send types?: asynchronous, unidirectional, synchronous, with implicit or explicit reply, eager reply (future), with possibly many replies, with priorities, with extra information (sender, arrival time...)?

- **activity**
  how does an object specify its acceptance of messages?, is it implicit (reactive objects) or explicit?, possibly computing several messages simultaneously (intra-object concurrency)?

- **synchronization and coordination**
  does an object control acceptance and activation of messages?, in regard of what ressources? (its state, current status of activations, requests...), and along which policy? (abstract states transition, guards...).

Proposal for a Unified Modeling Platform

We therefore propose a platform for modeling/simulating various OOCP models, languages and constructs within a single unified programming environment.

It provides a framework to help classify, design, reuse, and combine various OOCP strategies.

Motivations and Objectives

- **pedagogy**
  to help analyze and classify various OOCP models.
- **design**
  to help designing new OOCP strategies by derivation and combination of existing formalisms and strategies.
- **experiment**
  to provide an environment for active experiment with various OOCP models.

Actalk

Actalk is a platform based on the Smalltalk-80 programming environment designed with these goals in mind.

(*Actalk* name stands for *act*ive objects (and *actors*) in Small*talk*-80.)

Actalk is based on a kernel which describes the basic semantics of active objects (that is *reactive serialized* objects communicating through *asynchronous unidirectional* message passing).

This kernel may be extended *(and has actually been extended)* to simulate various OOCP programming languages models and constructs.

3 Design

Goals

- **uniformity** and **modularity**: one unique formalism.
- **minimality**: a *minimal kernel* for the simplest model of OOCP (reactive serialized active objects and asynchronous unidirectional message send).
- **extensibility** and **expressivity**: the kernel is extended (*subclassed*) in order to model various OOCP models, languages, and constructs.
- **simplicity**: we intend to represent fundamental characteristics and constructs of various OOCP languages. We don't address syntax considerations (Smalltalk-80 syntax model is used). We also usually don't address optimization, provability and security considerations.
- **integration**: Actalk is integrated within the Smalltalk-80 programming environment, allowing combination and reuse of standard Smalltalk-80 objects and programming environment tools.

By grouping various OOCP formalisms within a single unified programming environment, the Actalk platform eases analysis, comparison, reuse and combination of these formalisms.

Representation Model

We need to find good tradeoffs between these various goals, that is mainly between *simplicity*, *expressivity*, and *practicality*.

We chose Smalltalk-80 as the programming foundation and environment for its following qualities:

simplicity, high-levelness, modularity, flexibility, environment richness, and large availability.

Smalltalk-80 is modular and flexible enough to allow a very good integration of Actalk within its environment.

Smalltalk-80 also provides all basic objects (structures, functionalities and resources) needed to represent/implement:

- *active objects*: objects, classes, methods,
- their *communication*: messages, shared (message) queues,
- their *activities*: message computation (perform), processes,
- and their *synchronization*: shared (message) queues, semaphores.

Representation (Meta-)Level

Smalltalk-80 standard objects offer a high-level representation for Actalk active objects.

Because Actalk and OOCP are very close to (and well integrated within) Smalltalk-80 and OOP, this representation is actually very close to some reflective meta-description, while remaining concise and efficient.

Note that a fully reflective version/extension of Actalk has also been developed by Sylvain Giroux, and named ReActalk.

We don't follow this approach for the current platform in order to keep optimal decomposition/concision and efficiency. Sufficient flexibility is offered by the parameterization of the Actalk architecture, by it high-level implementation, and by its integration with the underlying Smalltalk-80 system.

4 Previous Results

Actually initial development of Actalk started in 1988.

- **Pedagogy**
  Simulation of Actor computation model behavior replacement, and ABCL/1 three types of message passing [ECOOP'89]. Variations and many examples.

  Simulations of POOL, CSP and OCCAM by a team of students at University of Nantes (advised by Jean Bézivin and Olivier Roux).

- **Environment**
  Focus quickly moved to the programming environment aspects.

  User-interface framework for active objects (extension of standard Smalltalk-80 MVC framework) [TOOLS-USA'91], including some specific interface generator.

  Generic scheduler of processes/activities [WOOC'93], plus scheduling visualization tools.

  (Most of these environment tools were developed by Loïc Lescaudron.)

- **Developments**
  Various DAI platforms (fine and large-grain) at Laforia, Paris.

  Also: simulation of multiprocessor communications, concurrent software engineering processes...

- **Experiments**
  - a reflective extension of Actalk: ReActalk (Sylvain Giroux).
  - study of exception handling mechanisms for OOCP.
  - strategies for concurrent constraint resolution.
  - compilation of production rules into concurrently activated daemons.

- **Parallelism**
  Distributed implementation of the Actalk kernel on top of GnuSmalltalk for a Transputer-based multiprocessor. (Unfortunately no time to complete it into some workable system.)

5 Recent Developments

Recently (mid-February 94) we restarted working on Actalk with this shared experience in mind.

Our goals were:

- to integrate and introduce **many more extensions** simulating various OOCP languages and constructs,
- to further improve the Actalk architecture by **increasing its modularity and expressivity**,
- to port Actalk onto the **latest Smalltalk-80 version** (4.1).

This has resulted in a large number of extensions and a finer decomposition of the platform parameterization.

Current Extensions and Simulations

model or language	*message send*	*activity*	*constructs*	*synchronization*
default	asynchronous & unidirectional	reactive & serialized	no	no
Actors	asynchronous & unidirectional	reactive & serialized	behavior replacement	behavior replacement
ABCL/1	3 types: now, past, future & 2 modes: standard, express	reactive & serialized	wait for, sender, atomic (no express), non resume	wait for (with where constraint)
POOL2	synchronous, public routines	autonomous body & serialized	answer (serve), unblocking answer, post-actions	answer
ConcurrentEiffel	asynchronous, future	body (live routine) & serialized	unblocking serve	unblocking serve
Concurrent Smalltalk-II	asynchronous, synchronous, future	reactive & serialized	relinquish, post-actions	relinquish
ACT++	asynchronous, future	reactive & serialized	abstract states transition	abstract states
OCore	asynchronous, future (q-structure)	reactive & serialized	user-defined events	user defined events and meta-level
abstract states	any	any	no	abstract states
guards	any	serialized	no	guards
guards & synchronization counters	any	concurrent	no	guards
generic invocations	any	concurrent	no	guards on generic invocations
mixed	any	concurrent	no	abstract states and guards

Examples

The environment also includes some relatively large set of small examples, mostly:

- *numerical:* factorial, fibonacci, prime numbers,
- *non numerical:* quick sort, distributed symbol table, behavior simulation,
- *synchronization:* bounded buffer, semaphore, printer queue, dining philosophers, readers & writers, with variations on concurrency, message ordering preservation, priority, fairness...

Several of these examples (e.g., bounded buffer including several inheritance anomaly examples) are derived in many different language models and synchronization policies in order to compare their relative pros and cons.

6 Architecture Decomposition

Actalk is decomposed into:

- the kernel which defines the basic semantics of active objects,
- and its various extensions, specified as class libraries representing and simulating various OOCP models, languages, constructs, and examples.

Decomposition and Parameterization of the Kernel

We identified three classes and a related set of methods.

Some of these methods are named *parameter methods* (or virtual methods) as they are designed to be redefined in subclasses in order to represent various OOCP models.

We look for a good balance between *modularity* as well as *concision* of the kernel.

Kernel Classes

Actalk kernel is decomposed in three kernel classes:

- `ActiveObject`: the behavior of the active object, that is the one which ultimately computes the messages.

  This class represents user programs as well as the program constructs defined by a specific OOCP language.

- `Activity`: the internal activity of the active object.

  This class provides the autonomy (process) to the active object. It also defines the way messages are selected, scheduled and handled. This includes possible synchronization policies on activation of messages.

- `Address`: 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 messages will be interpreted, that is possible types of communication.

This decomposition in three classes allowes:

- to decouple the semantics of transmission of messages with the semantics of their computation.
- to decouple user programs with the model of activity.
- to separate one program (active object behavior) with the specification of its inner synchronization (activity).
- one may associate several activities to a same address, thus simulating immediately the Actor model of computation.

7 Definition, Structure and Use of an Active Object

Definition and Use

- The programmer defines the behavior of an active object as a subclass of kernel class `ActiveObject` or one of its subclasses.
- An instance of this class represents the *behavior* of an active object.
- In order to actually *create and activate the active object*, one need to send the message `active` to the behavior.
- This implicitly creates and initializes the two other components, that is the activity and the address, of the active object.
- The *new address is returned* as the value of the method `active`.

Structure

The active object may be viewed with three successive layers:

- the address, which is the external reference to the active object,
- the internal activity which controls selection, acceptance and activation of requests,
- the inner behavior which ultimately computes the request.

Way it Works

The basic way an active object (standard type as defined by the kernel) works is as following:

- the address receives the message and enqueues it onto its private mailbox (mail queue),
- independently (eventually), the activity picks up the message and accepts it, that is delegates its computation to the behavior,
- the behavior performs the message.

8 Parameterization of Kernel Classes

The following three tables summarize for each kernel class its parameter methods and examples of redefinitions.

Class ActiveObject Parameter Methods

method selector	*parameter*	*default value*	*examples of redefinitions*
`privateInitialize`	*initialization*	nothing	none (yet!)
`active`	*start the activity*	create activity and address, start activity	(`SimulatedBodyObject`) send an initialization message to start the body
`activityClass`	*activity class*	`Activity`	(`PoolObject`) `PoolActivity`
`addressClass`	*address class*	`Address`	(`PoolObject`) `PoolAddress`

Class Activity Parameter Methods

method selector	*parameter*	*default value*	*examples of redefinitions*
`privateInitialize`	*initialization*	nothing	(`CountersActivity`) initialize synchronisation counters
`start`	*start the activity (process)*	start the process created by `createProcess`	(`Abcl3Activity`) start a second activity process specific to express messages
`createProcess`	*create the activity process*	create a process computing `body`	(`Basic2Activity`) create a handle to the process (useful for termination control)
`body`	*specification of the activity*	serially accept successive messages	(`SingleMessageActivity`) accept a single message (e.g., Actors behavior activity)
`nextMessage`	*next message to be accepted*	return and remove first message from message queue	(`EnabledSelectorsActivity`) return and remove first message whose selector is enabled
`acceptMessage:`	*accept and compute a message*	`performMessage:`	(`ConcurrentActivity`) start a subprocess to compute the message
`performMessage:`	*perform a message*	delegates actual perform to the behavior	(`ImplicitReplyActivity`) return the value to the implicit reply destination
`addressClass`	*address class*	`Address`	(`ImplicitReplyActivity`) `ImplicitReplyAddress`
`invocationClassFor:`	*invocation class*	`Message`	(`WithSenderActivity`) `WithSenderInvocation` (includes the sender)

Class Address Parameter Methods

method selector	*parameter*	*default value*	*examples of redefinitions*
`privateInitialize`	*initialization*	nothing	(`InvocationAddress`) initialize arrival time stamp counter
`receiveMessage:`	*receive a message*	`asynchronousSend: inMessageQueue:`	(`GenericSendAddress`) dispatch along the message send type and mode
`asynchronousSend: inMessageQueue:`	*receive an asynchronous send message*	enqueue message to message queue	(`SynchroConcurrentAddress`) atomically trigger the message reception event

9 Modular Decomposition and Parameterization

In order to possibly increase the expressivity of the platform without increasing complexity (number of parameter methods) and decreasing efficiency at the top of the hierarchy, we introduce further decomposition and parameterization *when* and *where* needed, that is within the hierarchy.

Kernel Levels

The kernel is functionally decomposed along three levels according to functionalities offered:

1. essential characteristics: the parameter methods and other basic methods,
2. extended functionalities, including control of the activity (e.g., termination) and *generic event methods*,
3. entry point for the user (kernel classes: `ActiveObject`, `Activity` and `Address`), with user-level facilities (tracing, checking, cleaning...).

Generic Event Methods

Level 2 introduces *generic event methods* associated to the three following events:

- *receive*: the active object receives a message.
- *accept*: the active object accepts a message and starts computing it.
- *complete*: the active object completes computation of the message.

Each generic method takes current message as an argument.

These methods may be used by the user to attach actions to a given class of active objects, e.g., to:

- trace activites,
- step computation,
- control scheduling of activities,...

These generic methods are also useful for modeling extensions, e.g., computing post actions for the POOL2 language (class `PoolActivity`).

In order to avoid confusion and shadowing between user-level (e.g., trace) and modeling-level (e.g., simulate post actions), we make a distinction between *user event methods* and *kernel event methods*.

Class `SynchroConcurrentActivity` introduces a third level: *synchronization event methods* which ensure atomicity of events.

Other examples of further decomposition within the hierarchy may be found with the following classes: `GenericSendAddress`, `SelectiveAcceptActivity`, `ConcurrentActivity`. They are described in the following tables summarizing respective hierarchies of each kernel class.

Hierarchy of Active Object Classes

model	*active object class*	*default activity/address class*
kernel	ActiveObject	Activity
ABCL/1 wait for	Abcl1Object	Abcl1Address
sender, wait for with where constraint	Abcl2Object	Abcl2Activity
express message control (atomic, non resume)	Abcl3Object	Abcl3Activity
Actors behavior replacement	ActorObject	SingleMessageActivity
explicit acceptance of a message (answer, unblocking answer)	ExplicitAcceptObject	ExplicitAcceptActivity
POOL2 post actions, public routines	PoolObject	PoolActivity
ConcurrentSmalltalk-II method suspension (relinquish)	SuspendObject	SuspendActivity
post actions	ConcurrentSmalltalkObject	ConcurrentSmalltalkActivity
reference to sender	WithSenderObject	WithSenderActivity

Hierarchy of Activity Classes

model	*activity class*	*default address class*
kernel	Activity	Address
reference to sender	WithSenderActivity	WithSenderAddress
ABCL/1 express message handling	Abcl3Activity	Abcl3Address
accept a single message	SingleMessageActivity	
implicit reply handling	ImplicitReplyActivity	ImplicitReplyAddress
enabled sets of selectors abstract states	EnabledSelectorsActivity AbstractStatesActivity	
explicit acceptance of messages	ExplicitAcceptActivity	
POOL2 post actions, public routines	PoolActivity	PoolAddress
selective acceptance of messages	SelectiveAcceptActivity	
guards	GuardsActivity	
concurrent activations, management of activity subprocesses	ConcurrentActivity	
method suspension post actions	SuspendActivity ConcurrentSmalltalkActivity	
synchronization events	SynchroConcurrentActivity	SynchroConcurrentAddress
guards with synchronization counters	CountersActivity	
generic notion of message invocation	InvocationActivity PlainInvocationActivity	InvocationAddress
mixed models	ASIActivity ASCActivity	

Hierarchy of Address Classes

model	*address class*
kernel	Address
Actors behavior replacement message handling (for external computation of behavior replacement)	ExternalReplaceActorAddress
generic dispatch of message send type and mode	GenericSendAddress
ABCL/1 past, now and future message send types reference to sender express message send mode ABCL/f single assignment future	Abcl1Address Abcl2Address Abcl3Address AbclfAddress
implicit reply handling	ImplicitReplyAddress
POOL2 synchronous message passing	PoolAddress
synchronization of concurrent activities: atomic triggering of message receive event	SynchroConcurrentAddress
generic invocations: increment the arrival time stamp counter	InvocationAddress
reference to sender	WithSenderAddress

Generic Management of Messages

Class `MailBox` (subclass of standard class `SharedQueue`) groups various accessing and filtering methods to access messages in a message queue (e.g., to look for some matching condition, or for a specific message pattern).

Class `Invocation` (subclass of standard class `Message`) provides generic invocations which may include or/and compute extra information (e.g., arrival time in order to ensure preservation message ordering, record number times skipped in order to ensure no starvation. . .).

10 The Combination Issue

Parameter methods *activityClass* and *addressClass* define default classes of activity and address.

One may redefine them for a given class, or/and also specify them when creating a specific active object.

Thus, one may combine between various active object, activity and address classes, to create various *hybrid* models.

Example: an active object conforming to Actor model of computation (classes `ActorObject` and `SingleMessageActivity`) *and* with the ABCL/1 three types of message passing (class `Abcl1Address`).

This example is valid, but some others would not work if there are some disjoint assumptions about their respective related classes. E.g., the activity class `Abcl3Activity` assumes that the address class defines an express message mailbox (as defined by class `Abcl3Address`).

Actalk provides a simple way to specify compatibility constraints between component classes and to check these compatibilities.

The designer or user may define the following methods: `activeObjectConstraint`, `activityConstraint` and `addressConstraint` to specify compatibility constraints for each component class.

11 Example: Combining Synchronization Policies

Abstract States

```
ImplicitReplyActivity subclass: #EnabledSelectorsActivity
   instanceVariableNames: 'enabledSelectors '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Ext-SelectActivity'

nextMessage
   ^self mailBox firstMessageWithCondition: [:message |
      enabledSelectors includes: message selector]


EnabledSelectorsActivity subclass: #AbstractStatesActivity
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Synchro-AbsStates'

"Compute the initial set of enabled selectors."
privateInitialize
   super privateInitialize.
   enabledSelectors := self perform: self initialAbstractState

"State transition: compute next set of enabled selectors."
kernelEventComplete: aMessage
   super kernelEventComplete: aMessage.
   enabledSelectors := self perform:
      (self nextAbstractStateAfter: aMessage selector)
```

Guards and Synchronization Counters

```
SynchroConcurrentActivity subclass: #CountersActivity
   instanceVariableNames: 'receivedDictionary acceptedDictionary completedDictionary '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Synchro-Counters'

privateInitialize
   super privateInitialize.
   self makeSynchroCounterDictionariesOnSelectors: oself class allScriptSelectors

"Checking acceptance of a message."
isSynchroAcceptableMessage: aMessage
   ^self satisfyGuardSelector: aMessage selector arguments: aMessage arguments

satisfyGuardSelector: selector arguments: argumentsArray
   ^self perform: (self findGuardSelector: selector) withArguments: argumentsArray

"Representation of guards: methods prefixed by symbol guardOF."
findGuardSelector: selector
   ^('guardOF' , selector) asSymbol

"Associate synchronization counters update to events."
synchroEventAccept: aMessage
   super synchroEventAccept: aMessage.
   self incrAccepted: aMessage selector

synchroEventComplete: aMessage
   super synchroEventComplete: aMessage.
   self incrCompleted: aMessage selector

synchroEventReceive: aMessage
   super synchroEventReceive: aMessage.
   self incrReceived: aMessage selector
```

```
"Creation of the synchronization counters."
makeSynchroCounterDictionariesOnSelectors: selectors
   receivedDictionary := IdentityDictionary new.
   acceptedDictionary := IdentityDictionary new.
   completedDictionary := IdentityDictionary new.
   selectors do: [:selector |
      receivedDictionary at: selector put: 0.
      acceptedDictionary at: selector put: 0.
      completedDictionary at: selector put: 0]

"Status of invocations."
accepted: selector
   ^acceptedDictionary at: selector

completed: selector
   ^completedDictionary at: selector

current: selector
   ^(self accepted: selector) - (self completed: selector)

pending: selector
   ^(self received: selector) - (self accepted: selector)

received: selector
   ^receivedDictionary at: selector

"Updating the synchronization counters."
incrAccepted: selector
   acceptedDictionary at: selector
      put: (acceptedDictionary at: selector) + 1

incrCompleted: selector
   completedDictionary at: selector
      put: (completedDictionary at: selector) + 1

incrReceived: selector
   receivedDictionary at: selector
      put: (receivedDictionary at: selector) + 1
```

Mixed Model: Abstract States + Guards

The following *mixed* model [Thomas PARLE'92] uses:

- *abstract states* to specify *state synchronization*,
- *and guards* to specify *activation synchronization* conditions.

```
CountersActivity subclass: #ASCActivity
   instanceVariableNames: 'enabledSelectors '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Synchro-ASC'

"These two methods are hand-coded combinations/mixins."
privateInitialize
   super privateInitialize.
   enabledSelectors := self perform: self initialAbstractState

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

"This expresses the conjonction of acceptance conditions."
isSynchroAcceptableMessage: aMessage
   ^(enabledSelectors includes: aMessage selector)
      and: [super isSynchroAcceptableMessage: aMessage]
```

Example: Bounded Buffer

```
ASCActivity subclass: #ASCBoundedBufferActivity
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Synchro-ASC-Examples'

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

full
   ^#(get)

partial
   ^(self empty) + (self full)

initialAbstractState
   ^#empty

"Abstract states transition. (oself: the active object behavior itself)."
nextAbstractStateAfter: selector
   ^oself isEmpty
      ifTrue:
         [#empty]
      ifFalse:
         [oself isFull
            ifTrue: [#full]
            ifFalse: [#partial]]

"Guards: one get and one put: may proceed concurrently."
guardOFget
   ^(self current: #get) = 0

guardOFput: item
   ^(self current: #put:) = 0
```

12 Evaluation and Critique

Actalk eases analysis and comparison of existing OOCP systems.

Main objective/use remains pedagogical.

Meanwhile Actalk provides a framework which may help to design, reuse, and combine existing ones into new ones. This means Actalk may be used as a prototyping environment.

It cannot express any kind of OOCP model.

Meanwhile we believe that it achieves some constructive compromise between expressivity, simplicity, and practicality concerns.

This is just an implementation of existing models after all!

Actalk helps reusing and combining models and policies because all their simulations are organized along some common framework and architecture.

This is just a matter of classifying and reorganizing a hierarchy of classes when simulating/including new models.

Our experience shows that: the initial architecture framework (from 88) remained mostly intact; current extensions model some relatively wide area of OOCP systems; further decomposition and parameterization may be added within the hierarchy without changing the kernel.

We cannot express any kind of language syntax without changing the standard Smalltalk-80 parser.

Our focus is on semantics and constructs, not syntax. When in need, we choose (the naive and incomplete solution) to tag message selector strings.

This not a formal way of modeling things.

Various simulations may be experimented actively thanks to the Smalltalk-80 based programming environment.

Ultimate goal would be to offer some kind of shared library of OOCP models, constructs, and policies.

13 Related and Further Work

Related Work

- ConcurrentSmalltalk [Yokote & Tokoro OOPSLA'86&87] [Okamura & Tokoro TOOLS-Pacific'90]
- Simtalk [Bézivin OOPSLA'87]
- Coda [McAffer Reflection-Workshop-OOPSLA'93]
- Generic Actalk Scheduler [Lescaudron PhD'92] [Briot & Lescaudron Concurrency-Workshop-ECOOP'92] [Briot WOOC'93]
- Prototalk [Dony et al. OOPSLA'92]

Future Work

To get feedback on actual use by other teams for experiments, and applications, e.g., ongoing project on natural language based on a multi-agent/blackboard architecture.

More investigation, extensions, examples.

To port and improve programming environment tools developed for previous version.

Access

Documented version on:

anonymous ftp
`camille.is.s.u-tokyo.ac.jp`; `cd /pub/actalk/`

WWW/Mosaic
`http://web.yl.is.s.u-tokyo.ac.jp/members/briot/actalk/actalk.html`