First National Workshop on Control Architectures of Robots - April 6,7 2006 - Montpellier

An Asynchronous Reflection Model for Object-oriented Distributed Reactive Systems

Jacques Malenfant[1]

Université Pierre et Marie Curie-Paris6, UMR 7606,
8 rue du Capitaine Scott F-75015 Paris, France
CNRS, UMR 7606, 8 rue du Capitaine Scott F-75015 Paris, France
Jacques.Malenfant@lip6.fr

Abstract. Today's distributed and embedded systems challenge the traditional procedural approach to reflection. Central to this approach is the use of an "implements" relationship to realize the connection between the meta and the base level. This restricted view of reflection is inappropriate in distributed or embedded computing, where part of the system to reflect upon cannot be captured in an "implements" relationship, either because we lack a centralized state or an essential ingredient lies outside the system. We introduce a novel *asynchronous reflective model*, $\mathcal{ARM}$, where the connection between levels use an asynchronous publish/subscribe communication model. We show not only that this model is better suited to distributed and reactive systems, but that it also generalizes the possible forms of reflection by adopting and adapting to B. Smith's "*right combination of connection and detachment*" between the base and the metalevel. $\mathcal{ARM}$ is applied to the reflective control of modular robots, which dynamic physical reconfigurability must be paralleled by a software reconfigurability offered by reflection. $\mathcal{ARM}$ then uses reactive objects founded on the GALS approach (globally asynchronous, locally synchronous), which implements synchronization by future values. A hybrid deliberative/reactive framework inspired by intelligent robotic control systems is implemented using the $\mathcal{ARM}[\mathcal{GALS}]$ for Java platform.

Keywords: reflection, object-oriented systems, distributed systems, embedded & reactive systems, event-based computing, AI robotics.

1 Introduction

Today's distributed and embedded systems operate for long periods of time, while large variations in the level of available resources are observed. Sustaining an acceptable level of performance in such contexts requires dynamic adaptation of applications. Reflection [39] has been proposed both as a conceptual framework and as an architectural blueprint to achieve dynamic adaptation of applications, yet these new systems challenge the traditional procedural approach to reflection.

In this paper, we consider the problem of controlling modular robots. A modular robot is one that is made of a large number of homogeneous and simple robotic entities, which can be physically assembled and reconfigured during the mission. Examples of such robots are CONRO [38] and M-TRAN [48]. The key concept behind modular robotics is that the shape provides for the function. Modular robots are morphologically reconfigurable to adapt to their mission: open field motion, go over obstacles, motion within pipes or between close walls, etc. We currently participate in a modular robot project called MAAM (Molecule := Atom | Atom + Molecule) where modules called *atoms* are build from a spherical kernel to which six orthogonal legs are attached. Legs can move in a cone and they can bind to each others to form molecules (see Figure 1). Because the morphology of the robot defines its function, we claim that a parallel reconfigurability of the software controlling the robot is necessary to dynamically implement the control of the new function.

Modular robots are examples of distributed embedded systems for which we introduce a novel reflection model called *Asynchronous Reflection Model*, or $\mathcal{ARM}$. We apply this new model to the software reconfiguration and dynamic adaptation of MAAM atoms. $\mathcal{ARM}$ aims at breaking the limits of procedural reflection to apply to distributed or embedded systems. It is seeking for generality and genericity, being parameterized both by the kind of base level supporting entities

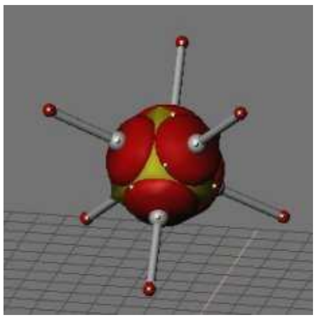
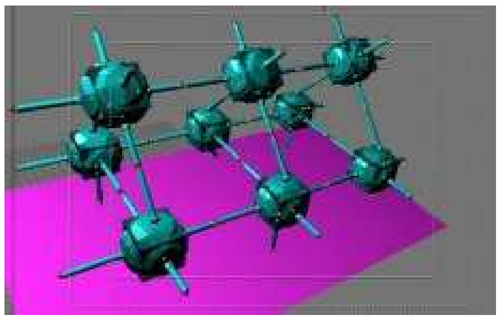



Fig. 1. The MAAM atom and molecule.

(components, active objects, reactive objects, and so on) and by the form of reified representation chosen to match the need for adaptation of the application. Hence, $\mathcal{ARM}$ should rather be seen as a generator $\mathcal{ARM}[\cdot](\cdot)$ of reflection models.

We also present a Java implementation of $\mathcal{ARM}$, which is based on an hybrid active and reactive object model. In this implementation, reactive objects use a synchronous approach to real-time [18], which meshes well with the event-based computation model of $\mathcal{ARM}$. However, we preserve active objects, better suited to program the metalevel entities. We therefore have adopted the globally asynchronous but locally synchronous (GALS) approach. The control of individual atoms is seen as a synchronous program, which can communicate with other atoms and the metalevel using asynchronous events. Our active and reactive objects implement synchronization using future values, a premiere in this context to our knowledge. For the MAAM project, we have developed a hybrid deliberative/reactive framework inspired from work in AI robotics [2], within the $\mathcal{ARM}[\mathcal{GALS}]$ for Java platform. A first deliberative metamodel for the dynamic adaptation of atoms has also been implemented.

The rest of the paper is organized as follows. In the next section, we introduce procedural reflection and then discuss its limitations in order to argue in favor of a novel asynchronous approach to reflection. In Section 3, we introduce the $\mathcal{ARM}$ model and its implementation in Java. Next, we address the problem of MAAM distributed real-time control by introducing our GALS model, its integration with $\mathcal{ARM}$ to give the $\mathcal{ARM}[\mathcal{GALS}]$ platform for Java, and its use to develop the deliberative/reactive framework used to program MAAM atoms. We then compare our approach to the related work and finally give conclusions and some perspectives of this work.

2 Motivations

2.1 Procedural reflection

Dating back to the seminal work of Smith, reflection is *"an entity's integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on and deals with its primary subject matter"* [40]. Reflective behavior is implemented by a metalevel, "*the most identifiable feature of reflective systems*", placed in a meta relationship with a base level. Although Smith did not impose any particular way to realize this relationship, his 3-Lisp language [39] and most of its descendents have adopted an "*implements*" relationship, where the metalevel interprets or otherwise processes the base level using traditional data structures of language processors reified into the language of the base level, thus enabling reflective computations.

The restriction of reflection to systems where the metalevel is in an "implements" relationship with the base level has been called *procedural reflection* by Jim des Rivières [14] and it has been defined as follows by Bobrow, Gabriel and White [5] in the context of reflective programming languages:

> *"Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation:*

introspection and intercession. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called reification.

To be more precise, reification encompasses both defining a representation (e.g. a class hierarchy) and obtaining objects at run-time that actually represent the current state of the computation. To be effective, introspection and intercession must operate on reified data that are continuously updated. *Causal connection* is the property of the link between the base and the metalevel imposing that any changes to one level leads to a causal effect on the other.

2.2 Limits of procedural reflection

The "implements" relationship exhibits many desirable properties, such as a full causal connection. When using metacircular interpreters as metalevel, it becomes easy to reify since everything is already represented by the metacircular interpreter as data structures in the base level language. However, it has the major drawback of introducing a full coupling between the two levels. Indeed, when adopting this kind of relationship, the meta in some sense *is* the base level, an observation that Danvy and Malmkjaer have formalized under the *single-threadedness* property of 3-Lisp like reflective languages [13].

The single-threadedness property says that, at any time, only one of the base or the metalevel is actually running. A usual corollary assumption permeating all reflective code is that nothing happens at the base level during reflective computations, and therefore modifications to the base level through its metalevel representation take effect before any computation steps can be carried over at the base level. In other words, metalevel and base level computations steps are *synchronous* with each other in the sense that they are totally ordered and happening in "mutual exclusion".

Except for some attempts in AI and agent-oriented programming, this view of reflection has permeated the vast majority of the reflective languages, middleware and systems proposed to date. Only a few recent reflective languages and middleware begin to timidly introduce alternatives (see §6). Unfortunately, this vision does not scale outside the traditional sequential programming field where it was first realized. Smith has often argued against such a restriction of reflection in an AI perspective, where his theory would apply to the relation between an intelligent entity and the world into which it operates [41]. Today, the challenge for this choice of the "implements" relationship to connect the base and metalevel comes from attempts to apply reflection into the distributed and embedded computing paradigms, where this relation fails to cope with the very nature of actual systems.

In distributed computing, the "implements" relationship goes against the absence of a global state and the inherent characteristics of a system made of independent computing nodes. As a result, most attempts to introduce reflection in distributed systems either restrict themselves to reflect upon individual (sequential) entities independently [45, 46, 30, 22, 28, 35, 47, 11, 31, 29], where a procedural approach can be applied, or reify only very particular aspects (e.g. message sending, stubs, ...) upon which local reflective computations can be introduced. Little work has been done to introduce reflection in embedded systems (however, see [21]); difficulties are similar to the ones while of AI applications foreseen by Smith, since they need reflection upon the "outside world" that indeed cannot be put in an "implements" relationship with the metalevel.

2.3 Advantages of an asynchronous non-implementing approach

To achieve its full generality, reflection must go beyond the traditional procedural approach and propose new ways to view reification and to implement the connection between the base level and its metalevel. In this paper, we propose that the publish/subscribe event-based computation model is better suited to implement the connection between the meta and the base level. We therefore introduce an *asynchronous reflective model* where the communication between levels uses asynchronous events and we claim that this model provides the means to adopt and adapt to the "*right combination of connection and detachment*" [39] necessary to reflect in general.

Of course, developing a reflective kernel upon a publish/subscribe middleware is not a real challenge. The crucial point concerns the revolution in the resulting form of reflection and therefore *what this new form allows us to do that was not possible in the traditional procedural reflection approach.* Generally speaking, using asynchronous events and distinguishing the metalevel representation from the data structures of the language processor go around the old discussions about the concurrency versus the non-concurrency between the base and the metalevels. Being relieved from its execution role, the metalevel can execute concurrently with the the base level.

More substantially, rejecting the strict synchrony imposed by procedural reflection to the benefit of a more finely-shaded semantics more or less synchronized, more or less fault-tolerant, or respecting different notions of causality between events, allows us to introduce a corresponding finely-shaded notion of causal connection. This corresponds exactly to what Smith was argueing for when requiring an equilibrium between connection and detachment among levels. The importance of this aspect appears clearly in real-time systems where strict deadlines must be met even when calling for reflective computations. These reflective computations must be sufficiently detached from the real-time base level to maintain the timeliness of the system.

Asynchronous reflection opens a wide spectrum of possible reified representation to be explored according to the form of introspection and intercession to be implemented. When relieved from the constraints of acting as the execution state of the language processor, reified representation can be defined as a *model*, in its very sense, i.e. an abstraction chosen for its proper goal. In the complex world of distributed embedded systems, it is illusive to hope for complete models, taking into account all aspects of the base level. In asynchronous reflection, incompleteness, fuzziness, or even randomness in representation can be smoothly integrated in models. Given the needs for adaptation, the model is defined by necessity, for the reified representation and for the means to construct the model, to instrospect and to intercede with the base level. A model is constructed by the metalevel by aggregation and processing of events received from the base level. The model needs not be unique. Models can easily be composite, thanks to the publish/subscribe technology.

Using its autonomous execution, the metalevel can also perceive events coming from other base level entities and even events coming from sensors collecting information from the non-computerized "outside" world, thus enabling truly distributed and embedded reflection. Autonomous execution also allows the metalevel to probe its environment to collect the necessary information. Unlike reflection *à la* 3-Lisp, the metalevel can take the lead in adaptation instead of waiting for requests from the base level.

The flexibility of publish/subscribe communication can also be recruited to provide a wide spectrum of connection/detachment possibilities. Events coming from the base level entity associated to the metalevel can be followed with a finer granularity than the ones coming from other entities or from the sensors. This can be done using the content-based filtering capabilities of message-oriented middleware. For example, to reflect upon the tactics and strategies of a robot football player, the metalevel need not be informed of the precise state of all the mechanisms controlling the movement of the robot. Moreover, in asynchronous procedural reflection, the grain of observable computational steps can be organized hierarchically (bigger steps being an aggregation of finer steps) so that the choice of notification granularity can be made on a per entity basis. This provides exactly the kind of tradeoffs Smith was looking for when he was talking about "*the right combination of connection and detachment*".

Furthermore, as filters can be modified dynamically, the connection/detachment tradeoffs need not be decided once and for all but rather adapted to the current situation. For example, at some point in time, the metalevel may want to ignore the level of remaining energy in the robot batteries, and inhibit the related events. But when the energy level crosses some threshold, the metalevel can switch into a mode where such events are sollicited and taken into account to adapt the behavior of the base level (robot). More generally, some sort of meta-events can mark thresholds crossing leading to a modification in the granularity of observation from the metalevel.

Finally, the very nature of reified representations will drastically change in this new settings. The experience we got when applying reflection to adapt systems dynamically [27], as well as the

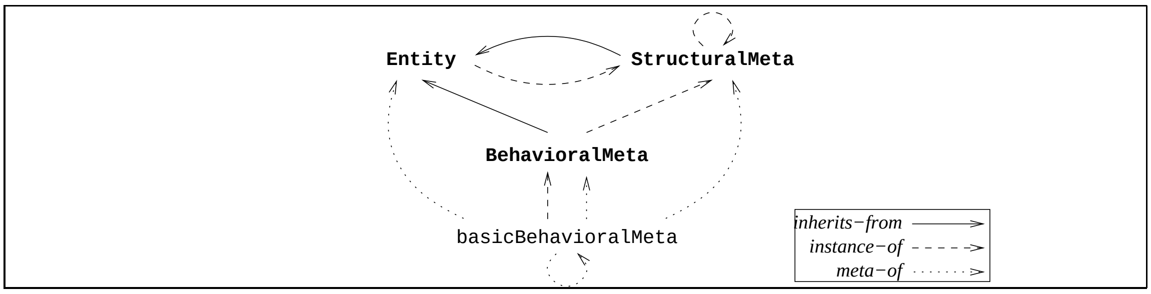


Fig. 2. Kernel entities of $\mathcal{ARM}$.

work in multi-agent systems show the necessity of prevision and planning for intercession. To adapt an application to the level of physical resources available is a control problem in the sense of classical control theory. Poor control policies can severely deteriorate the performance. To repetitively adapt to rapidly varying physical parameters can lead to a form of trashing where the system does nothing else but adapt itself. This well-known problem in control theory can be tackled by an appropriate choice of metalevel representation upon which viable policies, if not optimal, can be computed and then applied when interceding with the base level. Our asynchronous reflection model therefore goes towards a marriage of reflection and control to succeed in the dynamic adaptation of applications.

Of course, the major disadvantage of our asynchronous approach to reflection is the loss in reactiveness of the metalevel. Very fine-grained adaptations, to the level of instructions in the base level program, will not be efficiently implementable in the asynchronous approach. There are two counter-arguments to this. First, the kind of adaptability needed in distributed and embedded systems has often to do with variations of resources or context that happen infrequently compared to the rythm of instruction scheduling and execution but frequently compared to the duration of the whole program execution (sometimes years of continuous execution for some embedded systems). Second, nothing prevents a dual model, where a more traditional procedural reflection tightly integrated with the base level for language-oriented adaptation combines with an asynchronous reflection metalevel catering for environmental adaptation.

As a matter of fact, asynchronous reflection does not abolish procedural reflection, because intercession with a base level program still needs a reification of the program to be effective. Full procedural reification is not always necessary for all kinds of adaptation however. Reflective middleware such as reflective virtual machines can provide the necessary APIs to adapt the base level. In Java, for example, the possiblity to modify the code by hotswap as provided by the Java Platform Debugging Architecture (JPDA) [23] or code manipulation capabilities provided by reflective JIT compilers [34] can give enough flexibility to attack a wide spectrum of adaptation problems.

3 The Asynchronous Reflection Model

3.1 Kernel entities

The kernel of $\mathcal{ARM}$ is largely inspired from the ObjVlisp model of Cointe and Briot [7, 12], to which are added behavioral meta-objects in the line of Ferber [15]. The kernel is therefore built around three core structural meta-entities, `Entity`, `StructuralMeta` and `BehavioralMeta`, and a first behavioral meta-entity, hereafter called `basicBehavioralMeta`, which role is to be the behavioral meta-entity of all kernel entities, i.e. the three structural meta-entities ("classes") and itself. We have avoided the classical names `Object`, `Class` and `MetaObject` in order to emphasize the fact that $\mathcal{ARM}$ extends the procedural reflection of ObjVlisp with a more generic model for the reified representation of a computational entity from which many different specific representations can be developed.

Accordingly, we will use the word entity instead of object. $\mathcal{ARM}$ can be applied to several different kinds of base level entities: sequential objects, active objects, reactive objects, components, etc. To emphasize this genericity, we will use the notation $\mathcal{ARM}[\cdot]$ to introduce the fact that $\mathcal{ARM}$ is rather a generator of reflective kernels for given choices of the kind of base level entities. $\mathcal{ARM}$ need not be directive but rather liberal in the way it can be extended to apply to specific contexts and to implement specific applications. It therefore focuses more on APIs and relationships between entities than their actual content.

The figure 2 puts on the main "inheritance", "instantiation" and "meta-of" relationships of the $\mathcal{ARM}$ kernel. These relationships must be understood with a specific semantics in the context of $\mathcal{ARM}$, which can have nothing to do with their traditional meaning. Being instance here must be understood as "having as structural meta-entity", while inheritance must be understood as extending a structural meta-entity and the "meta-of" relationship as "having as behavioral meta-entity". Specific kernels generated from $\mathcal{ARM}$ with a given reified representation are responsible for giving a precise semantics to these relationships. In a procedural kernel, for example, they would take back their traditional meanings.

The graph of Figure 2 subsumes the one of ObjVlisp. `Entity` describes the common structure and behaviors of entities, while `StructuralMeta` describes the structure and behaviors common to all structural meta entities. Being an itself entity, `StructuralMeta` inherits from `Entity`. Being itself the first structural meta-entity, it is constructed in such a way that it is its own instance, i.e. it possesses the structure and behaviors of a structural meta-entity.

Adding to the relationships isomorphic to those of ObjVlisp, we have everything that have to do with `BehavioralMeta`. `BehavioralMeta` is the structural meta-entity for all behavioral meta-entities, and therefore is an instance of `StructuralMeta`. Being also an entity, it inherits from `Entity`. The first behavioral meta-entity, `basicBehavioralMeta`, is instance of `BehavioralMeta` and it is the behavioral meta-entity of all kernel entities, including itself.

The main flow of events that appear between the three different kinds of entities in order to implement the causal connection between the two levels are the following:

– notification of state changes or other semantically important modifications from the base level entity to its behavioral meta-entity allows the latter to construct or refine its model of what is currently going on at the base level;
– request for reflective computations can flow from the base level entity to its behavioral meta-entity, often to initiate an adaptation phase;
– requests from the behavioral meta-entity to the base level occur when adaptation are made;
– adaptation can also lead to modify the structural description of a base level entity, and therefore we have events flowing from the behavioral meta-entity to the structural one to implement these modifications;
– accordingly, structural modifications can lead to events flowing from the structural meta-entity to the base level object to harmonize the structure of the latter with the description held by the former.

Besides the requests from the behavioral to the structural meta-entities, all other communication flows go across level boundaries, and are therefore comparable to the classical procedural reflection operations "reify" and "reflect". Hence, the problem of designation of reified entities versus base level entities is posed. This aspect needs a more thorough study with the foundations of $\mathcal{ARM}$ to which we plan to return in future work.

3.2 Generalization of the reified representation concept

The reified representation is central to the metalevel. It is generally decomposed into a structural part and a behavioral part. The interest of this decomposition is that the structural part can often be shared among several base level entities (aka classes for objects), while the behavioral part, which accounts for the run-time state of base level entities, is not sharable by its very nature.

To get some point of reference, in sequential procedural reflection, the reified representation is implemented by classes (well-known) and by behavioral meta-objects. This representation comprises a description of the structure of base level entities, by a list of instance variables (with their types and other modifiers) and a description of the behavior by a list of methods that can be applied by the base level entities. The behavioral representation comprises the execution state of the base level entities, typically the current continuation and the current environment (given that the continuation embodies the environments of all subcomputations waiting still for a result of a method call). In distributed procedural reflection, the local behavioral representation comprises elements used to manage the concurrency such as the queue of incoming messages and threads.

Being open to several choices in representation that can even live together in one application, $\mathcal{ARM}$ is conceptually a generator of specific reflective models, noted $\mathcal{ARM}[\cdot](RR)$, given a choice RR of reified representation. The kernel defines a set of abstract "classes" for the reified representation that impose its constraints. $\mathcal{ARM}$ can therefore be concretely viewed as a model parameterized by a set of concrete classes derived from the representation abstract classes.

The design of this set of abstract classes results from an induction process aiming at making what transcends different representations appear. In this process, we have currently looked at three different forms of representation: one based on a classical procedural approach $\mathcal{ARM}[\cdot](\mathcal{P})$, one based on a deterministic finite-state automaton approach $\mathcal{ARM}[\cdot](\mathcal{DFA})$, and finally one based on a statechart approach $\mathcal{ARM}[\cdot](\mathcal{SC})$ where it is possible to have several levels of granularity in a clear semantic framework. The analysis of these three approaches has lead to the following minimal concepts of a reified representation:

– the *set of possible states* of the base level entity,
– a *behavior* that a base level object can apply to go from one state to another,
– the *set of behaviors* that a base level entity can apply,
– the *activation* of a base level object, and
– a possible *state* for a base level object.

The first three concepts are comprised in the structural part of the reified representation, while the activation concept is the hearth of the behavioral part as it will gather all the information about the run-time properties of an entity. The state concept can give us an account for the current state of an entity, thus being part of the behavioral part. On the other hand, the set of possible states can sometimes be defined in extension as the set of all possible individual states, hence leading to see the state concept as part of the structural representation too.

$\mathcal{ARM}$ represents these concepts as the five entities `State`, `StateSpace`, `Behavior`, `Behaviors` and `Activation`. For the sake of homogeneity, these can be considered as entities of the same kind as base level entities, but we do not impose that, as we will see in the $\mathcal{ARM}$ for Java platforms where they are abstract classes (partially) describing plain Java objects. The figure 4 provides the UML model of the kernel entities and reified representation for the $\mathcal{ARM}$ for Java platform.

These concepts map easily to different choices of reified representation. For example, the state of an entity can be a vector of instance variable values (for traditional objects), a state in an automaton (for the DFA approach), or a (possibly composite) state in the statechart approach. The set of possible states can be the cartesian product of set of admissible values (product type) for traditional objects, or sets of states defined in extensions for DFA or statcharts. A behavior can be seen as a method in traditional objects, but also as a transition in a DFA or a statechart. Accordingly, the set of behaviors can be either a method dictionary (procedural reflection) or the set of all transitions in the automaton (DFA or statechart).

3.3 Events of the communication protocol

The communication protocol between levels implies the use of asynchronous events. Besides events that implements more traditional method invocations, $\mathcal{ARM}$ also needs events to notify the metalevel of changes at the base level. One can imagine two general well-known approaches to notifying the metalevel. First, the base level can notify its state changes. Be it a differential description of the new state from the current one, this can lead to quite large events if the state of the entity comprises a large amount of data. A second approach notifies the actions taken at the base level, from which the metalevel model can reconstruct the new state given the current state.

These two notification modes are equally interesting in our context. The notification of actions can use a small amount of data if the parameters are of primitive data types only. On the other

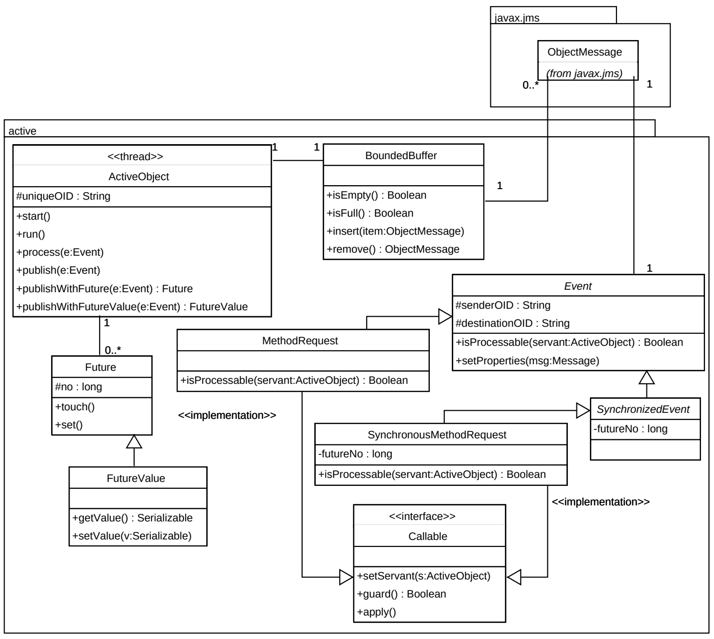


Fig. 3. Active objects.

hand, reconstructing a new state can be computationally intensive in some cases. The notification of state changes can be data intensive if it entails the communication of a large amount of data, but has a low computational cost. From another point of view, in distributed settings, notification of state changes is much more robust to loss of events than action notification. To let users choose the most appropriate form of notification for individual entities, $\mathcal{ARM}$ provides both a `StateEvent` generic state notification event and an `ActionEvent` generic action notification event.

4 A Java implementation of $\mathcal{ARM}$ for active objects

4.1 Asynchronous active objects as base level entities

A concurrent and distributed declension of $\mathcal{ARM}$ is implemented in the Java J2EE platform upon the basis of active objects communicating with asynchronous events and synchronizing using future values. In fact, the base level computational model upon which $\mathcal{ARM}$ for Java is founded is borrowed from Nierstrasz's Hybrid language [33]. Asynchronous active objects (AAO) represent the unit of concurrent and distributed computation, around which islands of unshared passive objects aggregate. Passive objects cannot communicate directly with objects (passive or active) that are not part of their island; they have to use their proprietary active object to do so. The implementation is inspired from the design pattern Active Object formalized by Schmidt [37].

The figure 3 gives a UML class diagram of our package `active`. The core functionality is implemented by the class `ActiveObject`, which is a thread (inherits from `Thread`) and which implements distributed message passing and notification using the Java Message Service (JMS) asynchronous communication API. All events used by $\mathcal{ARM}$ inherits from the class `Event`. An event has a sender and a destination; while queued by the receiver, a boolean method `isProcessable` tells whether the event is processable given the current state of the servant object.

Events can either carry data or activate methods in receivers. Method requests are executable events that implement the interface `Callable`. A callable event has a guard method which tests the processability of the event given the state of the servant. The servant is set by `setServant` upon enqueue of the event. When processable, the event become candidate to a dequeue and is applied using the method `apply`.

A method request can be unsynchronized (`MethodRequest`) or synchronized (`Synchronized-MethodRequest`). Synchronization is implemented using futures or promises [20, 25], a well-known synchronization method in asynchronous communication. When a synchronized event is published, an instance of `FutureValue` is created to represent the return value (resp. an instance of `Future` representing the return signal, when there is no value but just a synchronization signal to be sent back) in the sender. When the event has been processed, the receiver returns the result (resp. the signal) to the sender. When the sender tries to access the result (or the signal) using the `getValue` (resp. `touch`) method, there are two possibilities. If the value (resp. the signal) has already been received, the sender gets that value (resp. that signal) and continue its execution. If not received yet, it waits for the value (resp. signal) and resumes its execution only when the value arrives.

Events sent to active objects are queued into the object bounded buffer (class `BoundedBuffer`). The basic behavior of an active object is to repeatedly remove a processable event from its bounded buffer, and to process it. When none of the events are processable, or when there is no awaiting event at all, the active object becomes dormant until a processable event comes in.

To send events through JMS, active objects first put them into an instance of `javax.jms.ObjectMessage` and then send them using the JMS API. Communication in JMS is organized in topics. The package `active` uses the topic called `"active/Future"` to deliver future values (or signal) between active objects. Three other JMS topics are introduced to organize the communication in $\mathcal{ARM}$: `"arm/Entity"` for the communication between entities, `"arm/SMeta"` for the one between base level entities and their structural meta-entity, and `"arm/BMeta"` for the one between base level entities and their behavioral meta-entity.

4.2 $\mathcal{ARM}$ for Java kernel

The figure 4 gives a UML class diagram for the kernel entities and for the representation abstract classes of $\mathcal{ARM}[\mathcal{AAO}](\cdot)$ for Java. Besides the fact that `Entity` inherits from the class `ActiveObject` of the active package, we can identify relationships already defined in the model. Entities are represented by asynchronous active objects. Their main behavior appears in how they process incoming events, which is defined by the method `process`. A structural meta-entity provides you with methods to add (`addbehavior`), delete (`deleteBehavior`) or look up (`lookup`) behaviors. It also provides you with methods to get the initial state (`getInitialState`) for an instance of the structural meta, as well as a mapping from state events to states (`mapToState`) of their instances. A behavioral meta-entity provides you with a way to add a new base level entity to be the meta of (`addBaseLevelEntity`), and to delete an existing one (`deleteBaseLevelEntity`).

Because structural (and behavioral) meta-entities are also entities, they are also represented by asynchronous active objects. A bootstrap, implemented by the static method `bootstrap` of `StructuralMeta` creates the three corresponding entities for the kernel, as well as the basic behavioral meta-entity.

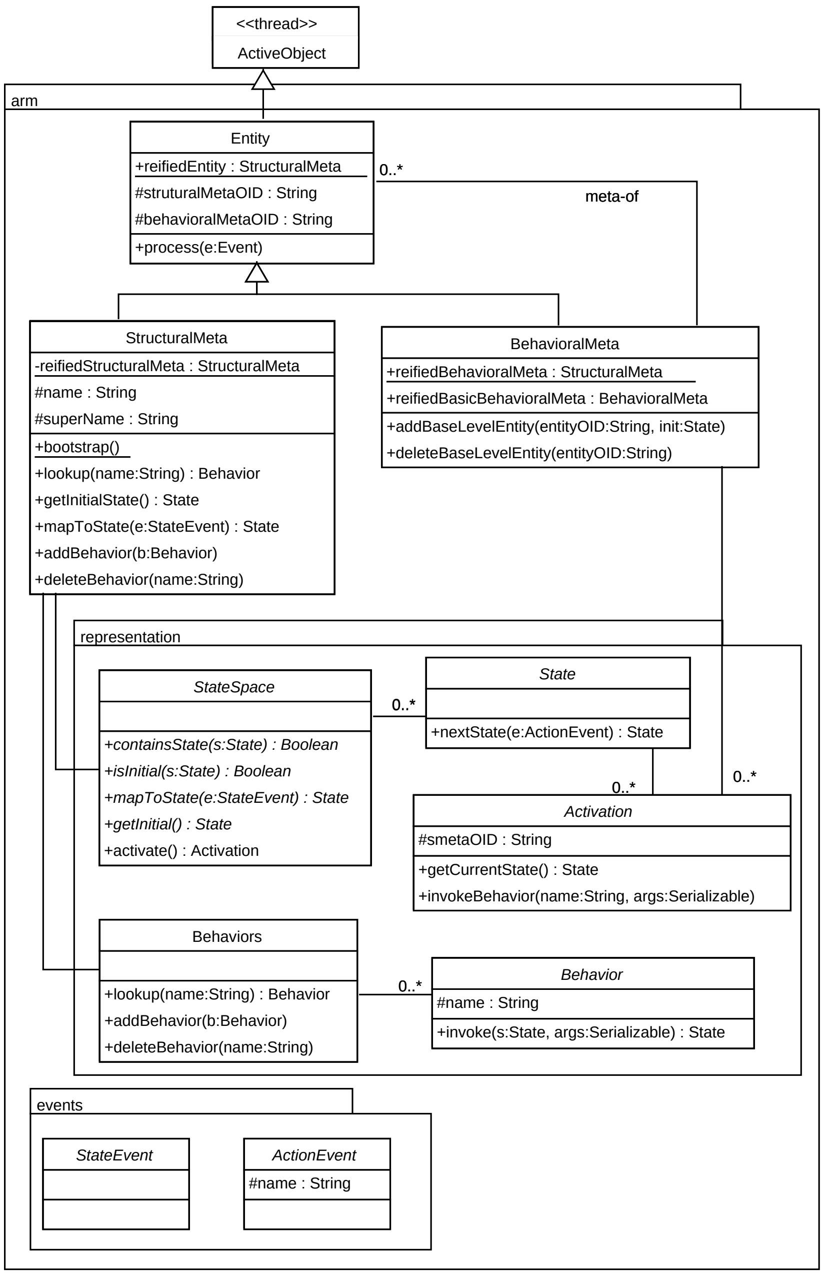
We have also defined a package `representation` containing the five classes corresponding to the representation entities in $\mathcal{ARM}$. These classes are abstract and minimal. Only the essential relationships between them are defined; further refinements are deferred to specifically generated kernels. Notice that an activation for an entity is obtained by calling the method `activate` defined on the `StateSpace` of the entity.

Finally, $\mathcal{ARM}$ defines the two abstract classes `ActionEvent` and `StateEvent`. An action event holds at least the name (the method name or the transition identifier) of the behavior which activation led to the notification of the event.

5 Application to AI robotics

5.1 Control architecture for AI robotics

AI robotic control systems are founded on the organization of the three basic primitive functions: sense, plan and act [32]. After a long domination from the hierarchical paradigm, where the emphasis is put on the creation of an exhaustive model of the environment used by planning, AI



<<thread>>
ActiveObject
arm
Entity
+reifiedEntity : StructuralMeta
#struturalMetaOID : String
#behavioralMetaOID : String
+process(e:Event)
0..*
meta-of
StructuralMeta
-reifiedStructuralMeta : StructuralMeta
#name : String
#superName : String
+bootstrap()
+lookup(name:String) : Behavior
+getInitialState() : State
+mapToState(e:StateEvent) : State
+addBehavior(b:Behavior)
+deleteBehavior(name:String)
BehavioralMeta
+reifiedBehavioralMeta : StructuralMeta
+reifiedBasicBehavioralMeta : BehavioralMeta
+addBaseLevelEntity(entityOID:String, init:State)
+deleteBaseLevelEntity(entityOID:String)
representation
StateSpace
+containsState(s:State) : Boolean
+isInitial(s:State) : Boolean
+mapToState(e:StateEvent) : State
+getInitial() : State
+activate() : Activation
0..*
State
+nextState(e:ActionEvent) : State
0..*
0..*
Activation
#smetaOID : String
+getCurrentState() : State
+invokeBehavior(name:String, args:Serializable)
Behaviors
+lookup(name:String) : Behavior
+addBehavior(b:Behavior)
+deleteBehavior(name:String)
0..*
Behavior
#name : String
+invoke(s:State, args:Serializable) : State
events
StateEvent
ActionEvent
#name : String

Fig. 4. Kernel and representation entities of $\mathcal{ARM}$.

robotics has been faced to the difficulty of creating such a complete model and to plan actions from such complex data structures within strict deadlines to cope with the real-time nature of robot operations.

Using lessons from ethology, Brooks [9] has proposed the reactive paradigm, which abandons planning in favor of very simple reflexes associating directly a reaction to each possible perceptions of the robot, without any memory of past perceptions and reactions. In this paradigm, the intelligence of the robot is emerging from the combination of a possibly large number of elementary reflexes. The absence of memory is grounded in Gibson's argument saying that "*the world is its own best representation*" [32, quoted in] which should be accessed only through perception. The

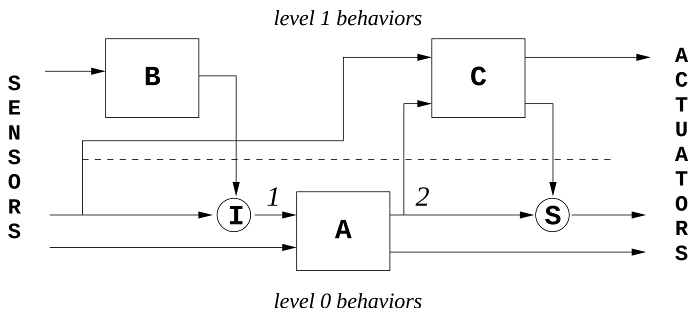


Fig. 5. Behaviors and subsumption: given perceptions, B can **I**nhibit the input 1 of A; similarly C can **S**uppress (replace) the output 2 of A.

reactive paradigm leads to the design of robots capable to react very rapidly to *stimuli* coming from their ecological niche. Reflex behaviors are implemented using three types of behavior modules:

1. perceptors, which role consists in reading sensors and producing *stimuli* (or the absence thereof) looked for by the robot (e.g. a spot of light in an image),
2. reactors, which role consists in computing the parameters of reactions given the actual *stimuli* and their intensity, and
3. actuators, which role is to transform the reaction parameters into orders to the physical actuators of the robot.

When composed, higher-level behaviors can inhibit lower-level behaviors, in much the same way our intelligent behaviors most of the time inhibit our animal ones. That's what is called subsumption in the reactive paradigm. To do that, modules are added to connect perceptors, reactors and actuators, which allow us to inhibit *stimuli* from perceptors or reaction parameters computed by rectors. The figure 5 illustrate these possibilities.

The successes of the reactive paradigm in the beginning of the nineties have given the first hope for operational situated intelligent robots. Unfortunately, the lack of a model of the robot environment and even more of planning soon appeared as a position far too extreme. Reactive control can become quite complex when trying to cope with some abnormal situations (looping, for instance), where planning could provide an answer. The emergent behavior of a large reactive robot can be very difficult to predict or to alter. Hybrid deliberative/reactive architectures have been introduced to get the best of both worlds. A reactive level takes care of robot reflexes in order to react in real-time to events from the environment, while a deliberative level can run in parallel to construct a model of the environment and to plan for future actions or to repair current faulty actions (such as looping). One possibility is to have the deliberative level producing new reactive programs to be used for a while at the reactive level until some goal has been reached or some abnormal situation is detected, and then to change for another reactive program.

5.2 A globally asynchronous but locally synchronous model

In our MAAM project, we aim at using reflection and $\mathcal{ARM}$ to implement the reactive part at the base level and the deliberative part at the metalevel. To do so, we have to provide a real-time computational model for the base level entities of ARM. Generally speaking, robotic systems are examples of the family of reactive systems. Reactive systems are those which main function is to react continuously to events occuring in their environment in order to produce reactions, often in the form of orders executed by actuators to act upon their environment. This distinguishes them from more traditional transformational systems, which compute outputs from inputs. Furthermore, reactive systems must cope with an environment which cannot wait, and they are generally intended to be deterministic. This distinguishes them from more general interactive systems. Typical reactive systems are plant control systems.

One of the most interesting approach to program reactive systems is the *synchronous* approach [18], which is based on a few simple hypothesis:

– events from the environment occur at discrete time instants,
– at each instant, the system computes reactions from all of the events perceived at that instant,
– the time to compute reactions is small compared to the time between two successive discrete instants, and
– reactions can lead to the emission of events, which are perceived instantaneously by all reactive processes at the next discrete time instant, along with environmental events.

The major advantages of the synchronous approach to reactive programming is the expressive power of synchronous parallel composition, the potential for efficient implementation, and the availability of formal verification methods [19]. The appropriateness of the synchronous programming approach to robot control justifies its choice for MAAM atoms. However, if synchronous programming is well adapted to the control of individual atoms, Halbwachs and Baghdadi [19] note that it is not so well adapted to the case of distributed embedded systems where the intrinsic asynchronism must necessarily be taken into account. Currently, researchers are looking at globally asynchronous but locally synchronous (GALS) architectures to cope with distributed embedded systems. In such architectures, local synchronous processes are composed with each others using asynchronous communication [19].

We have chosen the GALS approach for our MAAM atoms. Unfornatunately, if the synchronous approach matches very well the reactive part of MAAM atoms, it is not really appropriate to implement the more AI-oriented deliberative functions. Hence, the GALS model we have chosen for MAAM is a composition of both asynchronous active objects and synchronous reactive ones, composed using a publish/subscribe communication model. Schmidt and O'Ryan [36] have shown that publish/subscribe communication can have a level of performance that is compatible with distributed embedded programming, therefore justifying our choice.

The mix of active and reactive objects within one system raises the issue of synchronization mechanisms between both kind of entities. In the $\mathcal{ARM}$ for Java platform, we have used futures to implement synchronization between the asynchronous active objects. The mode of synchronization is not readily adoptable for reactive objects. Obviously, the waiting entailed by the traditional semantics of the `touch` and `getValue` operations is not appropriate for reactive objects given their real-time constraints. To solve the problem, we have simply noticed that active wait, which is usually considered as a bad practice in concurrent programming, is in fact the usual practice in real-time systems. Hence, we have adopted an active-wait semantics for reactive objects when synchronizing using futures. Futures are seen as any other *stimuli* upon which behavior modules can be fired (waiting). But if at some synchronous reactive cycle an awaited future is not available, other behaviors can continue their execution to keep up with the real-time deadlines while waiting for synchronization on other behaviors.

5.3 A Java implementation of $\mathcal{ARM}[\mathcal{GALS}]$

Most implementations of the synchronous approach re tightly integrated with synchronous languages, which are not appropriate to program robot deliberative functions. Few synchronous systems exist to date in Java[1], the language we have chosen for MAAM as a compromise between the real-time nature of the reactive level and the AI-bound nature of the deliberative level. We have therefore chosen to implement a minimal extension of our `active` package to introduce synchronous reactive objects with their semantics of synchronization on futures.

The figure 6 shows the new class diagram of the `active` package. In such an object-oriented settings, it would have been interesting to derive a class `ReactiveObject` from `ActiveObject`. Unfortunately, this would have caused difficulties when inheriting. Decoupling active and reactive behaviors would naturally lead to two classes in the $\mathcal{ARM}$ kernel: `Entity` and `ReactiveEntity`. Because of the single inheritance of Java, it is hard to satisfactorily implement the `ReactiveEntity` class because it would have to inherit both from `Entity` and from `ReactiveObject`. We have therefore chosen to keep just one class `ActiveObject` with two modes of operation chosen upon instantiation: synchronous and asynchronous.

[1] SugarCubes [43] is a notable exception, but its implementation is too resource consuming for the MAAM atom light-weight electronic.

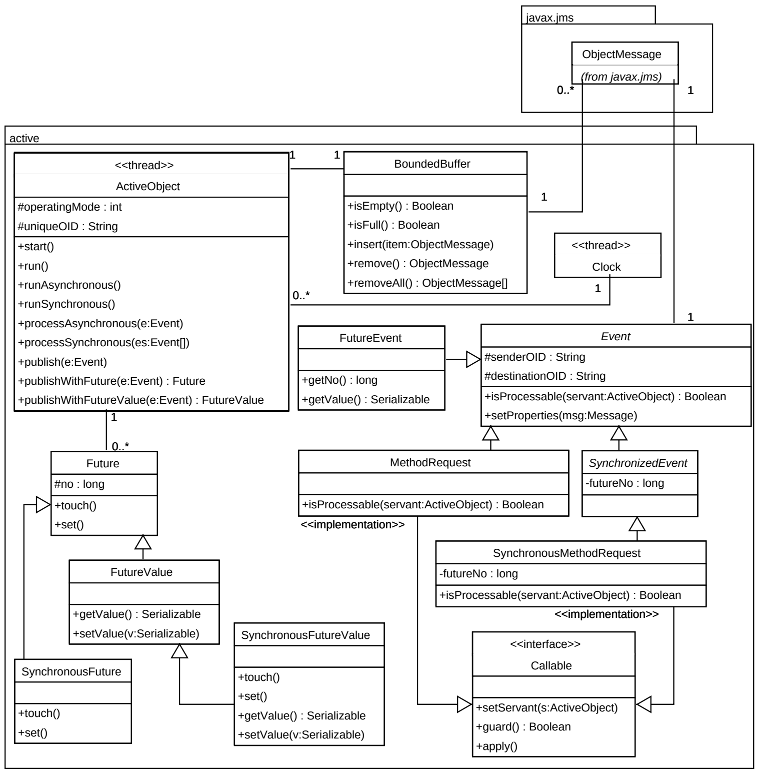


Fig. 6. Reactive objects.

The asynchronous mode of operation is the genuine $\mathcal{ARM}$ for Java behavior described in the preceding section. The synchronous mode is the one of reactive objects. In this mode, a clock object (class `Clock`) rythms the execution of reactive object threads. At each given period of time, the clock releases all the threads waiting. When released, the threads execute one reactive cycle, which consists in taking all events waiting in their bounded buffer and to process them according to their reactive behavior. When the processing is done, the threads put themselves on a wait for the next release signal from the clock. Futures are now considered as any other events for reactive objects, thus instance of the class `FutureEvent`. The classes `SynchronousFuture` and `SynchronousFutureValue` implement the active-wait semantics of futures for the synchronous mode of operation of reactive objects. Active objects keep the traditional semantics with `Future` and `FutureValue`

5.4 Hybrid reactive/deliberative model

In our MAAM atoms, robot control is defined as a combination of reactive schemas, themselves being sets of perceptor, reactor and actuator modules. From a programmer point of view, we have implemented a complete reactive framework under $\mathcal{ARM}[\mathcal{GALS}]$ for Java platform, which class diagram appears in Figure 8. The robot programmer has only to:

1. design his/her reactive schemas, taking possible subsumptions into account,
2. create the corresponding modules with classes inheriting from our framework classes,
3. create the classes for signals among modules by inheriting from our class `Signal`, and
4. create a class of reactive object, say `Robot`, inheriting from our class `AbstractRobot`, after which reactive behavior instances will have to be registered (using `addBehaviorModule`, `addConnector`, . . .).

To organize the reactive behaviors, we propose to have reactive schemas (`Schema`) composed of reactive modules (`ReactiveModule`), which themselves comprise behavioral modules (`BehaviorModule`). A reactive schema represents a logical function in the robot; it can be the basis for

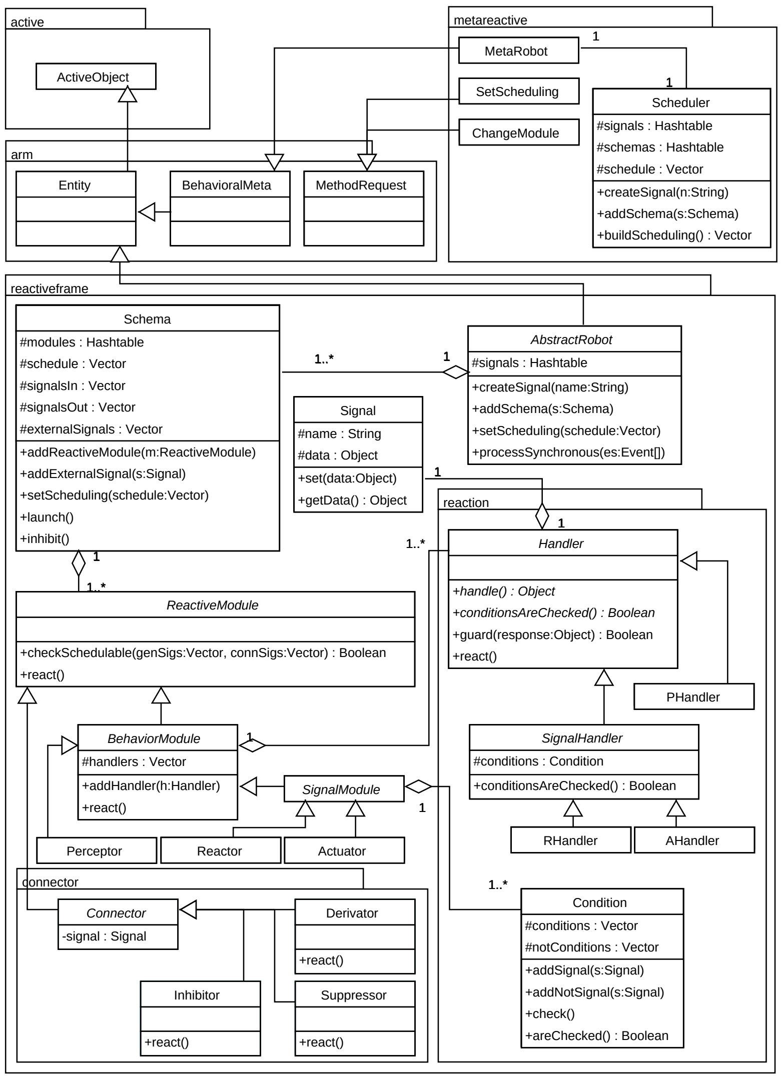


Fig. 7. Reactive framework under $\mathcal{ARM}[\mathcal{GALS}]$ for Java.

adaptation by dynamic addition or subtraction of schemas in the robot reactive program. Behavior modules can be perceptors (`Perceptor`), reactors (`Reactor`) and actuators (`Actuator`).

Schemas and behavior modules can be connected to each other using connectors (`Connector`), which comprise inhibitors (`Inhibitor`), suppressors (`Suppressor`) and derivators (`Derivator`). A schema must form an directed acyclic graph. The scheduling of behavior modules is implemented by a topological sort of that graph. Behavior modules in schemas are partially scheduled up to the connections to other schemas. The complete schedule of a robot instance is made when all schemas are known. When actually running, the robot simply executes all of its behavior modules, in turn, as scheduled within one cycle of synchronous reaction. A complete rescheduling is done when one or more schemas are added or subtracted by the robot metalevel. The deliberative part of the framework is currently implemented by one abstract class `MetaRobot`, which main purpose is to do the scheduling of its base level robot. The class `Scheduler` implements the above scheduling algorithm.

The figure 8 shows how our deliberative/reactive framework integrates with $\mathcal{ARM}[\mathcal{GALS}]$. The class `AbstractRobot` inherits from `Entity`, using the synchronous operating mode, from which base level robot entities are created (e.g. `robot1` and `robot2`). A class `BMetaRobot` defines behavioral

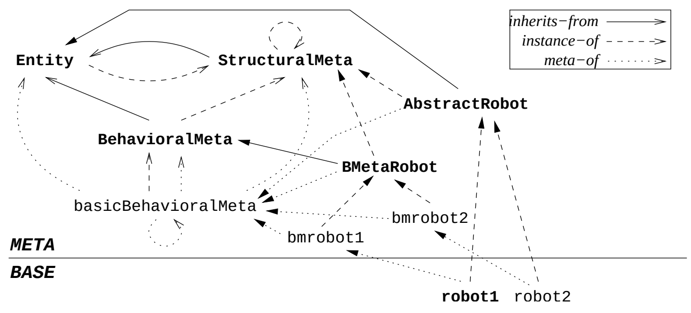


Fig. 8. MAAM robots under $\mathcal{ARM}[\mathcal{GALS}]$ for Java.

meta-entities for robot (e.g. `bmrobot1` and `bmrobot2`). The adaptation protocol put in place by `AbstractRobot` and `BMetaRobot` proceeds as follows:

1. at each synchronous cycle, notifications from the base level robots are sent to the behavioral meta,
2. the behavioral meta integrates these notifications into its reified model of the base level; this possibly fires an adaptation request sent to the base level as an event representing `setScheduling` or `changeModule` method request (the time needed to execute this adaptation must stay within the duration of a cycle to match the synchronous hypothesis),
3. the adaptation request is processed by the base level.

The adaptation request cannot be processed as other events in the synchronous processing cycle. Modifications are better handled when the base level is in some kind of renewal state, which needs the lowest possible amount of work to do the adaptation. For synchronous programs, this can either be at the beginning or at the end of a cycle. Depending on the priority to be given to adaptation compared to meeting the deadlines, the programmer can chose either policies.

6 Related work

In parallel with the industrial adoption of publish/subscribe communication for distributed programming, as the JMS API testifies, some work have introduced event-based communication in reflective operating systems and middleware [4, 3, 24, 44] and more generally in systems trying to implement forms of dynamic adaptability [27]. We should also mention work in computer-human interaction, like the MVC, as well as the Observer and State design patterns that have popularized the concept and the use of notification in general. This work, as well as their counterpart in reflection [6, 16] have inspired our work on $\mathcal{ARM}$.

Among reflective languages, LEAD++ [1] proposes an approach with tends towards $\mathcal{ARM}$ ideas, without breaking with procedural reflection though. The use of events has also inspired Dynascope [42], a supervision system, which Sosič read out as reflective. One can see in this system a precursor of the Java Platform Debugger Architecture (JPDA) [23]. MetaXa [17] uses events to reflect upon a Java-like virtual machine, but stays close to procedural reflection.

In object-oriented concurrent and distributed programming, most of the reflective languages restrict themselves to local reflection where the metaobject can be a processor for the base level (see [8] for a review of this wide area). Other systems have retricted themselves to the reification of very specific aspects of their implementation, such as stubs and proxies, upon which reflective computations can be locally implemented. All of these sticks to procedural reflection.

The actor and agent research community has identified and begins to explore ideas similar to the ones developed in $\mathcal{ARM}$. Without sharing the inspiration for agents, $\mathcal{ARM}$ can clearly share much of the implementation ideas with that area. Very few papers build bridges between reflection and control theory, Pii Lunau offering a notable exception [26], yet taking a procedural point of view.

The synchronous approach to reactive systes is generally associated with synchronous languages, like Esterel. The objective of a synchronous language is to offer a way to describe how events must be processed during a cycle of the logical clock. Several of these languages use the logical concurrent composition between computational activities an event emission. Robotic control architectures, like the deliberative/reactive ones pursue essentially the same goals. This is why we have not chosen to implement our base level entities using a synchronous language or system, like Rejo [43].

Arkin's book [2] is the reference in the area of reactive architectures in AI robotics. We did not find a reference implementation of these ideas in Java. In AI robotics, there is a tendency for everyone, that we unfortunately had to pursue, to reimplement his own framework. However, most of the work we had access to use a manual scheduling of behavior modules.

Halstead has proposed futures as a synchronization abstraction in MultiLisp [20], which has then been improved by Liskov and Shrira with promises [25]. Halbwachs and Baghdadi [19] propose to emulate different synchronization schemes in the synchronous approach, something we actually do with our implementation of futures for synchronous reactive systems. Caromel and Roudier [10] have proposed a reactive extension to the Eiffel// language, but they use an asynchronous approach to reactive systems borrowed from the Electre asynchronous reactive language.

7 Conclusion

$\mathcal{ARM}[\cdot](\cdot)$ is a new generic reflection model that we have argued in this paper to be much better suited to address the challenges of today's distributed and embedded systems. $\mathcal{ARM}$ is inspired from the ObjVlisp model to which behavioral meta entities are added. However, the traditional language processor role of the metalevel is abandonned in favor of a much more general role of model construction and controlled adaptation of the base level. The traditional procedural reified representation is also traded for the much more general concept of reification model, where incompleteness, fuzziness and probabilistic account of the base level can be used to capture the necessary properties of the base level to enable the wide-range of reflective capabilities needed in distributed and embedded systems.

$\mathcal{ARM}$ has been developed and applied to a modular robotics project called MAAM, where the physical reconfigurability of the robots has to be paralleled by an equivalent software reconfigurability. To that end, we have designed and implemented the $\mathcal{ARM}[\mathcal{GALS}]$ for Java platform. This platform uses a globally asynchronous but locally synchronous approach to the design of distributed embedded systems. In our platform, asynchronous active objects mesh with synchronous reactive objects to implement a hybrid deliberative/reactive framework typical in AI robotics. This implementation proposes to use futures for synchronization in GALS systems, a premiere to our knowledge.

Numerous perspectives are open by this work. Asynchronous reflection poses a large number of profound questions, such as the possibilities offered by new forms of reified representations, and the relationship between the kinds of adaptation and the required level of precision, synchronization, fault-tolerance and causality that must be imposed on events notifying state changes in the base level to the metalevel. Another important issue is the marriage of control theory and reflection that must be done to keep away from undesirable adaptation policies which would do nothing but repeatedly adapt the system to rapidly varying level of available physical resources for example. IBM has launched the autonomic computing initiative where such issues have a deep impact.

8 Acknowledgements

Simon Denier, student at the INSA of Rennes, has contributed to this work during his summer stay in 2002 and then during a semester research period in 2003. The MAAM project, led by Dominique Duhaut from the Université de Bretagne sud, is funded by the French CNRS under the ROBEA program.

References

1. N. Amano and T. Watanabe. An Approach for Constructing Dynamically Adaptable Component-Based Software Systems using LEAD++. In Cazzola, Stroud, and Tisato, eds, *Elec. Proceedings of the OOPSLA'99 Workshop on Object-Oriented Reflection and Software Engineering, OORaSE'99*, pages 1–16, 1999.
2. R. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.
3. G.S. Blair, A. Andersen, L. Blair, G. Coulson, and D. Sánchez. Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform. *IEE Proceedings — Software*, 147(1):13–21, February 2000.
4. G.S. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware'98*. IFIP, Springer-Verlag, 1998.
5. D.G. Bobrow, R.P. Gabriel, and J.L. White. CLOS in Context — The Shape of the Design Space. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS Perspective*, chapter 2, pages 29–61. MIT Press, 1993.
6. M. Braux and J. Noyé. Changement dynamique de comportement par composition de schmas de conception. In J. Malenfant and R. Rousseau, editors, *Proceedings of "Langages et Modèles à Objets, LMO'99"*, page ?? Hermès, 1999.
7. J.-P. Briot and P. Cointe. A Uniform Model for Object-Oriented Languages Using the Class Abstraction. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'87*, pages 40–43, 1987.
8. J.-P. Briot, R. Guerraoui, and K.P. Lohr. Concurrency and Distribution in Object-Oriented Programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.
9. R. A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.
10. D. Caromel and Y. Roudier. Reactive Programming in Eiffel//. In J.-P. Briot, J. M. Geib, and A. Yonezawa, editors, *Proceedings of the Conference on Object-Based Parallel and Distributed Computation*, pages 125–147. Springer-Verlag, 1996.
11. S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proceedings of ECOOP'93*, number 707 in Lecture Notes in Computer Science, pages 482–501. Springer-Verlag, July 1993.
12. P. Cointe. Metaclasses are First Class: the ObjVLisp Model. *Proceedings of OOPSLA'87, ACM Sigplan Notices*, 22(12):156–167, December 1987.
13. O. Danvy and K. Malmkjaer. Intensions and Extensions in a Reflective Tower. In *Proceedings of the ACM Symposium on Lisp and Functional Programming, LFP'88*, pages 327–341. ACM, 1988.
14. J. des Rivières and B. C. Smith. The implementation of procedurally reflective languages. In *Proceedings of the ACM Symposium on Lisp and Functional Programming, LFP'84*, pages 331–347. ACM, August 1984.
15. J. Ferber. Computational Reflection in Class Based Object-Oriented Languages. In *Proceedings of OOPSLA'89, ACM Sigplan Notices*, volume 24, pages 317–326, October 1989.
16. L.L Ferreira and C.M.F. Rubira. Reflective Design Patterns to Implement Fault Tolerance. In J.-C. Fabre and S. Chiba, editors, *Proceedings of the OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, pages 81–85. UTCCP Report 98-4, U. of Tsukuba, Center for Computational Physics, October 1998.
17. M. Golm and J. Kleinöder. MetaXa and the Future of Reflection. In J.-C. Fabre and S. Chiba, editors, *Proceedings of the OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, pages 1–5. UTCCP Report 98-4, University of Tsukuba, Center for Computational Physics, October 1998.
18. N. Halbwachs. Synchronous Programming of Reactive Systems – A Tutorial and Commented Bibliography. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of Computer Aided Verification, 10th International Conference, CAV'98*, number 1427 in LNCS, pages 1–16. Springer, 1998.
19. N. Halbwachs and S. Baghdadi. Synchronous Modelling of Asynchronous Systems. In A. L. Sangiovanni-Vincentelli and J. Sifakis, editors, *Proceedings of Embedded Software, Second International Conference, EMSOFT 2002*, number 2491 in LNCS, pages 240–251. Springer, October 2002.
20. R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transaction on Programming Languages and Systems*, 7(4):501–538, October 1985.
21. Y. Honda and M. Tokoro. Soft Real-Time Programming through Reflection. In A. Yonezawa and B. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture '92, Reflection and Meta-Level Architectures*, pages 12–23. RISE (Japan), ACM Sigplan, JSSST, IPSJ, November 1992.

22. Y. Ichisugi, S. Matsuoka, and A. Yonezawa. RbCl: A Reflective Object-Oriented Concurrent Language without a Run-Time Kernel. In A. Yonezawa and B. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture '92, Reflection and Meta-Level Architectures*, pages 24–35. RISE (Japan), ACM Sigplan, JSSST, IPSJ, November 1992.
23. Sun Microsystems, Java Platform Debugger Architecture Home Page. `http://java.sun.com/products/jpda`, 2002. SDK release 1.4.
24. F. Kon, M. Romàn, P. Liu, J. Mao, T. Yamane, L.C. Magalh aes, and R.H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In J. Sventek and G. Coulson, editors, *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware 2000*, number 1795 in LNCS, pages 121–143. Springer-Verlag, April 2000.
25. B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, PLDI'88*, pages 260–267. ACM Press, 1988.
26. C.P. Lunau. A Reflective Architecture for Process Control Applications. In *Proceedings of ECOOP'97*, number 1241 in LNCS, pages 170–189. Springer-Verlag, June 1997.
27. J. Malenfant, M.-T. Segarra, and F. André. Dynamic Adaptability: the MolèNE Experiment. In A. Yonezawa, editor, *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Reflection 2001*, volume 2192 of *LNCS*, pages 110–117. Springer-Verlag, September 2001.
28. H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, 27(10):127–144, October 1992.
29. H. Masuhara, S. Matsuoka, and A. Yonezawa. Implementing Parallel Language Constructs using a Reflective Object-Oriented Language. In G. Kiczales, editor, *Proceedings of the First International Conference on Reflection, Reflection'96*, pages 79–92. Xerox PARC, 1996.
30. S. Matsuoka, T. Watanabe, and A. Yonezewa. Hybrid Group Reflective Architecture for Object-Oriented Reflective Programming. In *Proceedings of ECOOP'91*, number 512 in LNCS, pages 231–250. Springer-Verlag, July 1991.
31. J. McAffer. Meta-Level Programming with CodA. In *Proceedings of ECOOP'95*, number 952 in LNCS, pages 190–214. Springer-Verlag, August 1995.
32. R.R. Murphy. *Introduction to AI Robotics.* MIT Press, 2000.
33. O. Nierstrasz. Active Objects in Hybrid. *Proceedings of OOPSLA'87, ACM Sigplan Notices*, 22(12):243–253, December 1987.
34. H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java. In E. Bertino, editor, *Proceedings of ECOOP 2000*, number 1850 in LNCS, pages 362–387. AITO, Springer-Verlag, 2000.
35. H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework. In A. Yonezawa and B. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture '92, Reflection and Meta-Level Architectures*, pages 36–47. RISE (Japan), ACM Sigplan, JSSST, IPSJ, November 1992.
36. D. C. Schmidt and C. O'Ryan. Patterns and Performance of Distributed Real-time and Embedded Publisher/Subscriber Architectures. *Journal of Systems and Software*, 66(3):213–223, June 2003.
37. D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* Wiley & Sons, 2000.
38. W.-M. Shen, B. Salemi, and P. Will. Hormone-Inspired Adaptative Communication and Distributed Control for CONRO Self-Reconfigurable Robots. In *IEEE Transactions on Robotics and Automation*, October 2002.
39. B.C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM, January 1984.
40. B.C. Smith. Discussion session. First Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA/ECOOP'90, October 1990.
41. B.C. Smith. What do you mean, *meta?* In J.-P. Briot, B. Foote, G. Kiczales, M.H. Ibrahim, S. Matsuoka, and T. Watanabe, editors, *Informal Proceedings of the First Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA/ECOOP'90*, October 1990.
42. R. Sosič. Introspective computer systems. *Electrotechnical Review*, 59(5):292–298, December 1992.
43. J.-F. Susini. Implementation de l'approche reactive en Java : les SugarCubes v2. In *Actes de Modélisation des Systèmes Réactifs, MSR'99.* Hermès, 1999.
44. N. Wang, M. Kircher, and D.C. Schmidt. Towards a Reflective Middleware Framework for QoS-Enabled CORBA Component Model Applications. In *Proceedings of the Reflective Middleware Workshop, RM 2000*, 2000. Electronic proceedings.

45. T. Watanabe and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. *Proceedings of OOPSLA'88, ACM Sigplan Notices*, 23(11):306–315, November 1988.
46. T. Watanabe and A. Yonezawa. An Actor-Based Metalevel Architecture for Group-Wide Reflection. In J.-P. Briot, B. Foote, G. Kiczales, M.H. Ibrahim, S. Matsuoka, and T. Watanabe, editors, *Informal Proceedings of the First Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA/ECOOP'90*, October 1990.
47. Y. Yokote. The Apertos Reflective Operating System: The Concept and its Implementation. *Proceedings OOPSLA'92, ACM SIGPLAN Notices*, 27(10):414–434, October 1992.
48. E. Yoshida, S. Murata, S. Kokaji, A. Kamimura, K. Tomita, and H. Kurokawa. Get Back In Shape! A Hardware Prototype Self-Reconfigurable Modular Microrobot that Uses Shape Memory Alloy. *IEEE Robotics & Automation Magazine*, 9(4):54–60, 2002.