# INTRODUCING REFLECTION IN ARCHITECTURE DESCRIPTION LANGUAGES*

Carlos E. Cuesta[1], Pablo de la Fuente[2], Manuel Barrio-Solórzano[3], and M. Encarnación Beato[4]

[1,2,3]*Departamento de Informática, Universidad de Valladolid, Spain*

[4]*Escuela Universitaria de Informática, Universidad Pontificia de Salamanca, Spain*

{cecuesta,pfuente,mbarrio}@infor.uva.es, ebeato@upsa.es

**Abstract** This document describes the structure of $\mathcal{P}i\mathcal{L}ar$, an Architectural Description Language based on concepts from the field of Reflection, following a proposal suggested in previous work. First, motivations and ideas behind its design are outlined. Next, the language is divided in two parts: a declarative Structural Language, which makes possible to define an architecture's static skeleton; and an imperative Dynamic Language, which appears as a set of constraining rules written in a concurrent language. Both languages are intertwined with the *reification* concept, which has a reflective origin. Its meaning and consequences are commented in detail. After this, the language's formal semantics are informally described; it is conceived as a system of concurrent processes, communicating by means of channels. It is argued that this semantics fits perfectly with architectural concepts. Finally, a solution for the classical problem of the *Dining Philosophers* is included as an example, to show how this ADL describes the dynamic evolution in a system. The paper concludes emphasizing the generality and usefulness of the language.

**Keywords:** Software Architecture, Reflection, Dynamic Architecture, Reification, MARMOL, $\mathcal{P}i\mathcal{L}ar$, Meta-Component.

## 1. Introduction

One of the most important tasks in Software Architecture is the definition of adequate languages (ADLs: Architecture Description Languages) for the specification of the structure of software systems. In later years, a certain number of them have been proposed (Allen, 1997; Canal et al., 1999; Luckham and Vera, 1995; Magee and Kramer, 1996), and there have been some discussion

*This work has been partially sponsored by the Spanish Commission for Science and Technology, through the CICYT Research Fund TEL99-0335-C04-04.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35607-5_15

J. Bosch et al. (eds.), *Software Architecture*

about the fundamental abstractions to consider in their design (Medvidovic and Taylor, 2000). No doubt, one of the most complex is the description of the dynamic aspects which define the *evolution* of an architecture.

In previous work (Cuesta et al., 2001; Cuesta et al., 2002) we have proposed the introduction in Software Architecture of concepts from the field of Computational Reflection (Maes, 1987), and the interest of such approach for the description of dynamic concerns has been justified. To show this, we have defined a reflective framework for architectural description (MARMOL), which is independent of any existing language.

However, the usefulness of such a (reflective) model can only be tested in practice by applying MARMOL to a concrete ADL. For this purpose, we have defined the language $\mathcal{P}i\mathcal{L}ar$ (***PiLar is a Language for architectural description***). This paper provides the first detailed presentation of this language; previously it just has been informally sketched (Cuesta et al., 2001) or summarized (Cuesta et al., 2002) to be used in more specific contexts.

We have designed $\mathcal{P}i\mathcal{L}ar$ using the MARMOL framework, but this has not been the only considered issue. On the contrary, we have tried to formulate it such that it allows for the natural specification of the widest possible range of systems, being a real alternative to other existing ADLs. The design has then used three basic ideas. First, to provide the language with most of the constructs present in other ADLs, to serve as a vehicle for integration. Second, to make it compatible with a formal semantics, inspired in previous works combining architectural abstractions and process algebras (Allen, 1997; Canal et al., 1999; Bernardo et al., 2001). And third, to avoid that the mixing of all these features –including reification– make it unnecessarily complex.

In the following sections we review the basic concepts of reflection, to later describe the syntax of our language, divided in a *Structural Language*, which describes the static skeleton of an architecture and is comparable to most ADLs, and a *Dynamic Language*, which provides the rules to make it evolve and change. Special attention is paid to the concept of *reification*, which provides the reflective structure, and concurrent semantics are briefly outlined. Finally, we expose a solution for the (small-grained) *Dining Philosophers* problem, which serves both as a concrete example and as a summary for some of the concepts discussed through the document.

## 2. Reflection in Architecture

The concept of Reflection has been used in many areas since its inception, such as artificial intelligence, object-orientation, programming or middleware systems. Recent research in Software Architecture (Cazzola et al., 1999; Cuesta et al., 2001) suggests that the field can also benefit from using this notion, which provides a way to manage many useful abstractions.

Reflection is defined as *"the capability of a system to reason and act upon itself"* (Maes, 1987). In short, it provides self-observation –*introspection*– and self-control –*intercession*–. The concept has many implications in pure and applied logic and programming, but here we will study it just inside Software Architecture. Previous work in the MARMOL framework (Cuesta et al., 2001) has already determined which notions are considered of interest to the field, and why are they important. We have used this theoretical basis to design $\mathcal{P}i\mathcal{L}ar$. The way it deals with reflection derives from Maes' adaptation of the concept to object-orientation (Maes, 1987).

The basic idea is simple: if an architecture is able to control and modify itself, obviously it shows dynamic capabilities. Using reflection as a structuring concept, we can identify which parts realize normal operation, and which ones *reflect* upon these, thus providing self-control –and then dynamism–. This way, there's no need to define special components (Allen et al., 1998; Le Métayer, 1998) or any other extra notion. Those ideas can be related to usual constructs by means of reflection, and everything can still be first-order.

Reflection always divides an architecture in two: the part which is controlled and the part which controls. These are respectively named *base-level* and *meta-level*. The meta-level can be described as the context or interpreter in which the base-level is defined; this provides a causal connection between them, expressed in MARMOL as a *reification* relationship[1].

Each level consists of a subsystem, which is also divided in components. From this perspective, a normal, base-level component is named an *avatar*, and it is related to –*reified by*– one or more *meta-components* in the meta-level. But components in the meta-level could be reified themselves, hence defining another level: a *meta-meta-level*. There's no limit to this process, and a specification can thus implicitly be divided in several meta-layers. Details about the resulting model can be obtained from (Cuesta et al., 2001).

Reflection uses a certain amount of ideas, but from an architectural perspective, we are only interested in relationships within the structure. Then, *reification* is the only concept we need to introduce: it provides support for everything else. Obviously, $\mathcal{P}i\mathcal{L}ar$ defines it, and in fact it appears as one of the central notions of the language. It is discussed in section 4.

## 3. Structural Language

There are two different concerns in any dynamic ADL: the description of the static structure and the characterization of patterns of evolution. Hence, to provide the necessary separation of concerns, we can consider $\mathcal{P}i\mathcal{L}ar$ syntax as divided in two parts: a *Structural Language*, which describes the static skeleton

[1] It is bidirectional: it fuses the usual *reflection* and *reification* operations from programming languages.

of systems, and a *Dynamic Language*, which defines the rules to make it change. Of course they use a lot of common notions –and terms–. A component definition consists then of a mandatory structural part, and an optional dynamic part.

There's just one kind of element: the **component**, defined as a basic compositional unit, encapsulated and defined by one or more segmented (non-atomic) interfaces, and present in a configuration through one or several *instances*. Hence it defines a type, and it is also known as *archtype*[2]. It may be either *primitive* or *composite*. In the first case, just the interface is described, as we aren't trying to describe the functionality but the interaction structure. As in Rapide (Luckham and Vera, 1995), it serves as a placeholder, and it is meant to refer to an existing implementation for a module.

In the second case, the composition of several mutually interacting instances is hidden behind the declared interface. From outside, there's no difference with a primitive component; then, it can be composed itself, shaping the *component hierarchy* which is typical of architectural description.

In $\mathcal{P}i\mathcal{L}ar$, a component definition has four parts, none of them strictly mandatory: **interfaces, configurations, reifications** and **constraints**. The latter two provide the reflective structure and the dynamic language, respectively, and they will be described in sections 4 and 5. An **interface** is a logically coherent aggregation of **ports**, which are in turn defined as the component's interaction points, expressing both services and requirements. A component may have one or more interfaces: they are simply put aside.

A **configuration** defines a *composite component*. It consists of a set of component **instances**, interacting through **bindings** or *attachments*, defining a complete subsystem. We describe both of them in the following.

To ease the explanation, in Figure 1 we provide an example: the classical pipeline of filters (Shaw and Garlan, 1996). The architecture simply takes a number of basic pieces, named *filters* –which are primitive, and thus we only describe their interface–, and put them together, connecting their inputs and outputs in cascade. *PipeIter* is an iterative version; *PipeRec* describes a recursive one. Their meaning should be obvious at the end of this section.

There are four kinds of **instance** declarations in $\mathcal{P}i\mathcal{L}ar$, namely **typed** instances, **arrays** of instances, **parameterized** components and **reified types**. The first is the most basic case: it defines a single instance of an archtype. For example, in *PipeRec*, *head* is declared as an instance of type *Filter*. The second is the usual **array** declaration of an indexed set of instances. For example, in *PipeIter*, $F$ is an $N$-sized array of *Filter* components. **Parameterized** components support for the definition of generic abstractions, by providing optional

[2]The term "component" has proven to be misleading, as it is used in different contexts to refer to types (the most correct) or to instances (the most usual). To avoid confusion, we will ever use *component* to mean *instance*, and *archtype* to mean *type*.

```
\component Filter (
 \interface (
   port left | port right ))

\component PipeIter <N:int> (
  \interface (
    port into | port out )
  \config (
    F[1.. N]:Filter |
    \bind (
      into = F[1]. left |
      out  = F[N]. right |
      \for (i: 2.. N−1)
        (F[i−1]. right =
                F[i] . left))
  ))
```

```
\component PipeRec <N:int> (
 \interface (
   port into | port out )
 \config (
   head:Filter |
   \ if (N>1)
     ( tail:PipeRec<N−1> |
       \bind (
         into = head.left |
         out  = tail .out |
         head.right = tail . into ))
   \else
     ( \bind (
         into = head.left |
         out  = head.right ))
 ))
```

*Figure 1.* A Pipeline of Filters: Iterative and Recursive Versions

arguments enclosed in angles[3]. Both *PipeIter* and *PipeRec* are parameterized components, and their argument $N$ is meant to give their size. Syntax for parameterized instances is similar: we can find an example in the declaration of *tail* at *PipeRec*. Finally, **reified types** refers to the use of types as instances in the meta-level, as explained in section 4.

On the other side, there are three kinds of **bindings** in $\mathcal{P}i\mathcal{L}ar$, namely **links**, and **hierarchical** and **typed** bindings. They are very closely related. **Links** are single attachments, describing a direct connection (communication) between two ports at the same level. **Hierarchical** bindings are nearly identical: they connect an internal port with a port at the composite's interface, thus *exporting* it. Both can be named or remain anonymous, and use the *equals* (=) sign to denote an attachment, as they can be seen as "fusing" the ports they join.

**Typed bindings** are meant to provide multiple ($n$-ary), complex connections. At the base level, they are considered just like any other attachment –wiring–. The only difference is that they are typed, to enable us to classify the interactions. The syntax to declare their types is similar to that of *primitive* components, defining their interface. This way, they can connect more than two ports. We use a component-like syntax because later we will use them as if they were components. Thus they have an explicit *name*, and are declared like component instances, but with an argument: the set of ports to connect.

[3]Note that parameters only have atomic types –integer numbers, for instance–, as we are not trying to introduce higher-order abstractions in the language.

Typed bindings are indeed quite similar to standard *connectors* (Shaw, 1994), but with some significant nuances. First, they are not related to components through attachments, as they are attachments themselves. This avoid strange, *intermediate* composites. Second, in the base level they don't define a behavior: it is provided at the meta-level by (implicit) reification. Third, they are then controlled by a meta-component; there's no need to define them as first-class notions, as they can be described as components at another abstraction level, which capture adequately the idea of interaction being carried "inside". Those *meta-level connectors* were described in (Cuesta et al., 2001), so we don't deal with them in detail here; but we mention them briefly in section 4.

The language has also support for *conditional* and *iterative* constructs, to simplify complex descriptions, like many other ADLs. There are some examples in Figure 1, marked by keywords **\if** and **\for**. As should be obvious by now, the language also admits *recursive* definitions.

## 4. Reification

*Reification* is the only reflective notion we need to introduce in the language, as stated in section 2. It's a structural concept, but it's also important for dynamism, as it defines how constraints will be combined.

In $\mathcal{P}i\mathcal{L}ar$, like in MARMOL, reification expresses a bidirectional relationship: it can be seen as a (privileged) link between an *avatar* (base-component) and a *meta-component*. This unifies the usual unidirectional operations: a meta-component has access to all the internal details of the avatar it *reflects* in; an avatar abstractions' are *reified* as meta-components. Hence a meta-component can freely alter an avatar.

Reification is a many-to-many relationship: a meta-component can reflect in many avatars; an avatar can be reified by many meta-components. Then, the base-level and the meta-level are connected by one or more reification links between the components they contain.

Components in the meta-level are generically known as *meta-level components*. We distinguish two kinds of them: those which directly reflect in an avatar (*meta-components*) and those which don't (termed simply *meta-level components*, again). But of course a meta-level component interacts freely with meta-components at its own level, thus it can also (indirectly) affect the behavior of avatars (Cuesta et al., 2001).

Reification in $\mathcal{P}i\mathcal{L}ar$ can be either *explicit* or *implicit*. Explicit reification uses a specific syntax, namely a **\reify** construct, both in structural definitions and dynamic constraints. It explicitly states that a component is a meta-component for another, typically the one which is defined. The link can receive a name, hence making it possible to modify it later.

Implicit reification simply consists of usual declarations: when an instance $C$ of an archtype $T$ is created, not only a component $C$, but also a *meta-component* $T$ is declared, together with a reification link between them. Hence, *every declared archtype can be used anytime as a meta-component*; then, it's easy to change the influence of type definitions within the system. Apart from that, implicit and explicit reification have exactly the same meaning.

Combined with primitives of the Dynamic Language (section 5) this provides the language with a great expressive power. The idea is that meta-components have access to internal details of avatars, overriding encapsulation; and they are able to manage them as if they were *data*. Dynamism, as many other interesting abstractions, is then easily provided.

We should briefly comment on typed bindings. As they are typed, they are also implicitly reified. Hence we can define complex meta-components which reify bindings, providing them with elaborate behaviors. Thus, the usual distinction between components and connectors (Shaw, 1994) is not strictly required, as we're using normal, *interacting* components, but placing them at the meta-level. As noted previously (section 2), these may be considered as *meta-level connectors* (Cuesta et al., 2001).

## 5. Dynamic Language

Behavior in $\mathcal{P}i\mathcal{L}ar$ is provided by a number of rules, described with the Dynamic Language, and scattered throughout component definitions in the **\constraint** section. They are then associated with an already defined structural skeleton, to ensure certain properties and react to several situations.

The main concern of the Dynamic Language is to specify *behavior*; this includes dynamism, but also communication protocols. For this reason, a process algebraic syntax[4] is a natural choice: constructs in the Dynamic Language are directly based in those of CCS (Milner, 1989). This allows us to make an easy integration with the $\pi$-calculus semantics, but maintaining simplicity from the user's perspective, who tends to consider mobile algebras rather difficult.

A **constraint** consists on one or several rule definitions, forming a modular subsystem. The first one defines how they are combined and triggered. They are analogous to CCS processes, and describe, basically, how interactions are managed. Definitions may be *recursive*, and this is indeed a standard way to express repetition. The other is *replication* (**\bang**), which launches a new copy of a (finite) process each time its first action is triggered.

There are two atomic actions: *sending* (*c!(x)*) and *receiving* (*c?(x)*) a message. They are similar to the corresponding operations in a process algebra, and we express them with the popular CSP syntax. The second one *waits* for a

[4] $\mathcal{P}i\mathcal{L}ar$ has an algebraic notation and also a programming-like syntax. We will review just the latter one.

*Table 1.* Some significant reflective primitives in $\mathcal{P}i\mathcal{L}ar$

| *Keyword* | *Notation* | *Informal Meaning* |
|---|---|---|
| **avatar** | $\alpha$ | Reference to the avatar to which the constraint is applied. |
| **self** | $\gamma$ | Reference to the component in which the constraint is defined. |
| **avatarSet** | $\Sigma\alpha$ | Set of all of the avatars reflected by this meta-component. |
| **portSet** | $\Sigma\pi(a)$ | Set of all the (public) ports in a component $a$ |
| **new** | $\nu(a:t)$ | Creates a new avatar $a$ of type $t$ (if specified). |
| **del** | $\delta a$ | Destroys (deletes) an entity (an avatar or link) $a$. |
| **reify** | $\rho N(a:m)$ | Creates a reification $N$ between avatar $a$ and meta-component $m$. |
| **findr** | $\phi N(a:m)$ | Finds a reification link between avatar $a$ and meta-component $m$. |

message, and thus is usually used as the *guard* (triggering event) of a process. Interaction points in a constraint are ports and internal channels: to access them hierarchically, we use the well-known *dot notation* (*Agent1.rd*).

Actions are combined by *parallel composition*, separated by the | symbol, or forming a *sequence*, indicated by a blank space, a carriage return or the ; symbol. Parenthesis can be freely used to avoid ambiguity. Control constructs from the structural language are also allowed, being even more important here. Hence we have convenient iterative (**for**) and conditional (**if**) constructs. There's also an additional conditional-like structure (**when**), designed to observe events without intercepting –consuming– messages.

Like in $\pi$-calculus, every communication is supported by channels, conceived as interaction locations, which provide the standard asynchronous handshake protocol. But unlike in the calculus, they can only be used in controlled ways. First, channels in a constraint are always *private*, except when they are specifically exported by declaring them as *ports* in the structural definition. Second, those ports only provide external interaction when they're bound. Thus *bindings* are seen as communication links, wiring two or more ports. In the case of simple bindings (*links* and hierarchical bindings), two ports are connected ($a = b$) by establishing a protocol such that data sent in one of them are received in the other, and vice versa. On the other hand, *typed bindings* manage a certain amount of ports, usually in complex ways, and will be reified. Thus in the end they will also be treated like a component definition: their constraints are translated, composing a process.

The basic Dynamic Language is just that: to end, we only need to consider the support for *reification*, which is provided by means of several reflective primitives, supported by this relationship. The most interesting among them are summarized and described in Table 1. When reading their definition, we should have in mind that these primitives are declared in an archtype, so they must be considered *inside* a meta-component, which limits their scope.

A constraint is enforced by a meta-component and must be obeyed by all of its avatars. It should be read as a rule situated at the meta-level: so **self** is the meta-component, and **avatar** is each one of the reflected components. Both meta-level and base-level behavior are then provided, such that they are closely related, even involving external interaction at each level. We can see also that any meta-component serves as a common reference to all of its avatars, so they can concurrently compete to access it.

## 5.1. Brief Note about Semantics

$\mathcal{P}i\mathcal{L}ar$ semantics conceives an architecture as a set of concurrent processes, communicating by means of named channels, in which each component is a process. We consider that concurrency is both natural and essential. We also believe that, linguistically, the key aspect in an architectural description is the management of *names*, a global term to include ports, instances, archtypes and links. For this reason, the $\pi$-calculus (Milner, 1999) has been the perfect tool to specify $\mathcal{P}i\mathcal{L}ar$'s formal semantics.

There's no space here to comment it in detail. We will just remark that there is a direct correlation between architectural and process-algebraic concepts. Composition and interaction are basic notions in both fields. Encapsulation is provided by in the $\pi$-calculus by name (scope) restriction.

Ports are simply unrestricted (public) channels –actually, pairs of channels–, and attachments are translated as (lightweight) processes in charge of communicating those channels. Configurations correspond to channel and scope topologies, and constraints are obviously translated as process definitions.

Reification is perhaps the most complex feature, but it has been tackled with a standard *interposition* mechanism. This resulted in an an elegant outcome: reification appears as equivalent to *superimposition*, a construct for concurrent supervised control (Katz, 1993). Indeed, this notion has been already introduced in Software Architecture, and also to provide dynamic capabilities (Wermelinger and Fiadeiro, 1999). Anyway, it was used just as a means to extend components, while in it is the key concept in our proposal, and is perhaps more naturally related to dynamism.

# 6. $\mathcal{P}i\mathcal{L}ar$: A Case Study

In this section we include a formulation for a possible solution for a variant of the classical problem known as *The Dining Philosophers*, in order to show a complete, non-trivial architectural description in $\mathcal{P}i\mathcal{L}ar$. We have chosen this example because it is well-known. Several formulations are possible, but the one we have used makes an intensive use of $\mathcal{P}i\mathcal{L}ar$'s Dynamic Language.

As frequently happens with many classical examples, strictly speaking our solution for this one is not describing a *good* architecture. Both components and

```
\component philosopher (
 ( port lhand | port rhand | port eat ))

\component fork (
 \interface ( port T )
 \constraint (
   P def= T?(get); T!(ok) ; Q
   Q def= T?(X); \if (X=get) (T!(no) ; Q)
                 \if (X=put) (P)))

\component group <n:int> (
 \interface ( port rhand | port lfork )
 \config (
   \if ( n=1 )
      ( aphil:philosopher | afork:fork |
        \bind ( aphil.rhand = rhand |
                afork.T = lfork ))
   \else ( aphil:philosopher | afork:fork |
           rest:group<n−1> |
           \bind ( rest.rhand = rhand |
                   afork.T = lfork ))
 \constraint (
   P def= \if ( n>1 )
           ( avatar.aphil.eat?()
             \if ( n mod 2 = 0)
                 ( Z; D; F ) % left−handed
             \else ( D; Z; F ))
                        % right−handed

   Z def= avatar.afork.T!(get)
          avatar.afork.T?(X)
          \if (X = ok)
            ( \bind (
                le: avatar.afork.T =
                    avatar.aphil.lhand))
          \else ( tau; Z )

   D def= avatar.rest.lfork!(get)
          avatar.rest.lfork?(X)
          \if (X = ok)
            ( \bind (
                ri: avatar.rest.lfork =
                    avatar.aphil.rhand)
          \else (tau ; D)

   F def= tau; le!(put) ; ri!(put)
          \del (le) ; \del (ri) ; P
 ))

\component system <n:int> (
  \config ( all:grupo<n> |
    \bind ( all.rhand = all.lfork )))

\base−level : ( system )
```

*Figure 2.* The Hurried Philosophers Problem

their ports have been chosen poorly, and their granularity is too small. However, it shows how the language modifies a system's communication topology.

As it is known, the problem outlines a community of $n$ philosophers which alternatively either *think* or *eat* spaghetti sitting around a (round) table. To eat each philosopher must use a fork on his left and another on his right; but as there are only $n$ forks, is not possible to have everybody eating simultaneously. A solution must provide a strategy for philosophers to share the forks.

The most frequent architectural presentation of the problem makes use of its circular nature. It describes two $n$-sized arrays of philosophers and forks, such that each philosopher is linked with his two adjacent forks. This is a very simple approach, but this is **not** the one in which we are interested.

Our solution uses quite a different approach. Here, bindings will be *created* when the philosopher gets (*get*) a fork, and *destroyed* when he frees it (*put*). Thus, the evolution of the system becomes even more radical, though it is not a very efficient solution to the problem. Moreover, to show more about the language's usage, we have decided to use a *recursive* organization (see Figure 2), instead of a (possibly easier) iterative equivalent.

```
\component system <n:int> (
   ( port add )
   \config ( all:group<n> |
     \bind ( all .rhand = all . lfork  ))
   \constraint (
     \relfy  R1 ( group :plus<n>(add) )))

\component plus <m:int> (
   ( port add )
   \constraint (
     \bang (add?();
     \if  ( avatar.n = m ) % for recursion
       ( (\new avatar.aphil:philosopher;
          \new avatar.afork:fork );
          \bind (
            ( avatar.aphil .rhand = avatar.rhand |
              avatar.afork.T = avatar. lfork) ;
            avatar.n = n+1; m = m+1 ))))
```

*Figure 3.* An Extension: The Evolving Philosophers Variant

For this purpose, we define an intermediate parameterized component, namely what we call a *group*. It describes a community of $n$ philosophers and forks, with a right hand (*rhand*) on its right side, and a fork (*lfork*) on its left side. This group grew up constructively, starting with the 1-sized group.

Each philosopher has (still) two hands (*lhand*, *rhand*) and joins the group bringing with him a single-handled fork (*T*). A 1-sized group (*group<1>*) is created when a philosopher sits at the table and places his fork at his left, without getting it. The sides of the block thus created are, conveniently, a right hand and a left fork. The group grows when a new pair philosopher-fork sits at the table, always positioning himself *at the left* of the existing group. The sides of this newly created group are again a left fork (the new one) and a right hand (still the first one). The consistency of the recursive definition is guaranteed by the interplay with the group parameter ($n$).

Finally, the *system* component just "closes the circle", by placing the last left fork within the reach of the first philosopher's right hand.

Among all the solutions the literature proposes for this problem, we have chosen the one known as *The Hurried Philosophers* (Lynch, 1996), which is quite simple. It consists of considering *left-handed* and *right-handed* philosophers, which here strictly means that they always get first their left (resp. right) fork, and then the other. It's easy to show that if there is at least one philosopher of each kind, *deadlock* is always avoided. In Figure 2 we have made each *even* philosopher right-handed, and each *odd* philosopher left-handed. Thus the system in the Figure maintains the condition, and the problem is solved.

The dynamic part of this example is provided by Figure 3, which describes a slight extension to the Hurried Philosophers' system, enabling the dynamic

addition of new philosophers to the group. This variant is usually known as *The Evolving Philosophers* problem (Kramer and Magee, 1990). Here our interest is just to show how to extend a system's behavior –providing even dynamism– by means of reification.

Only the *system* component is modified, to insert into it a new port *add*, provided to order the addition of new philosophers to the group. This behavior is inducted by means of a reification which modifies the *group* meta-component. This solution is quite complex, as we're modifying a parameterized, recursive component, and linking it to the new port. In a (sensible) array-like organization, it would have been much easier. It should be noted that the new philosopher and his fork are created *inside* the avatar –i.e., inside the group–; so the *already existing constraints* will be also applied to them. This ensures a consistent behaviour, even after changing the system.

## 7. Conclusions and Future Work

$\mathcal{P}i\mathcal{L}ar$ is a recent ADL and it is still in test phase; but in our experience it appears to be both useful and expressive. The examples in this paper are just giving a taste of its flexibility; it seems that almost any conceivable system could be described in several ways. The concept of *meta-component* makes possible to cope with many architectural abstractions in a common framework. For example, our notion of *reification* might be seen as a privileged form of control-driven coordination. In practice, this point of view could make the concept more easy to describe; for this reason, a comparison with control-driven models such as IWIM (Arbab, 1996) has been planned.

The development of $\mathcal{P}i\mathcal{L}ar$ continues. The formal definition of its semantics into the untyped $\pi$-calculus (Milner, 1999) is almost finished, and a consistent type system, based on polymorphic $\pi$-calculus, is being developed. Formal semantics will be useful in two ways. First, $\pi$-calculus provides support for (limited) semi-automated analysis (Victor and Moller, 1994), which makes possible to verify interaction properties and test several system's degree of equivalence. Second, the type system can serve a lot of purposes, as already suggested by (Bernardo et al., 2001), but the most obvious is to ensure the consistency of a description.

$\mathcal{P}i\mathcal{L}ar$ is conceived primarily as a language for specification, as we think that this is the main concern in Software Architecture; but plans for the future include a closer relationship to implementation. There are several languages –even some Java libraries– which could easily translate $\pi$-calculus protocols, providing a means to simulate the architecture's behaviour. It would be even possible to use this result as "glue" code in a real system, but this would be possibly inadequate, and that's not our purpose.

*PiLar* may seem difficult, but actually it is easier to use than a standard process algebra. It's complexity is comparable to that of other ADLs which specify behavior, like Wright or Rapide. Perhaps reification is an uneasy concept, but it only appears when it's strictly required, and when it does it always serves a purpose. In any case, the language offers a comfortable framework to explore a wide range of abstractions, and indeed it seems to be an useful tool for the description of dynamic structures.

## References


Allen, R. (1997). *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University. Technical report CMU-CS-97-144.

Allen, R., Douence, R., and Garlan, D. (1998). Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of 1998 Conference on Fundamental Approaches to Software Engineering*, Lisbon, Portugal.

Arbab, F. (1996). The IWIM Model for Coordination of Concurrent Activities. In Ciancarini, P. and Hankin, C., editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 24–56, Cesena, Italia. Springer Verlag.

Bernardo, M., Ciancarini, P., and Donatiello, L. (2001). Detecting Architectural Mismatches in Process Algebraic Descriptions of Software Systems. In Kazman, R., Kruchten, P., Verhoef, C., and van Vliet, H., editors, *Proceedings of the Second Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, Amsterdam. IEEE Computer Society Press.

Canal, C., Pimentel, E., and Troya, J. M. (1999). Specification and Refinement of Dynamic Software Architectures. In *Software Architecture*, pages 107–126. Kluwer Academic Publishing.

Cazzola, W., Savigni, A., Sosio, A., and Tisato, F. (1999). Architectural Reflection: Concepts, Design, and Evaluation. Technical Report RI-DSI 234-99, Univ. degli studi di Milano.

Cuesta, C. E., de la Fuente, P., Barrio-Solórzano, M., and Beato, E. (2001). Dynamic Coordination Architecture through the use of Reflection. In *Proceedings of 16th ACM Symposium on Applied Computing (SAC2001)*, pages 134–140, Las Vegas, NV. ACM Press.

Cuesta, C. E., de la Fuente, P., Barrio-Solórzano, M., and Beato, E. (2002). Coordination in a Reflective Architecture Description Language. In Arbab, F. and Talcott, C., editors, *Coordination Models and Languages*, volume 2315 of *Lecture Notes in Computer Science*, pages 141–148, York, UK. Springer Verlag.

Katz, S. (1993). A Superimposition Control Construct for Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356.

Kramer, J. and Magee, J. (1990). The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16.

Le Métayer, D. (1998). Describing Software Architecture Styles Using Graph Grammars. *IEEE Transactions on Software Engineering*, 24(7):521–553.

Luckham, D. C. and Vera, J. (1995). An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann.

Maes, P. (1987). Concepts and Experiments in Computational Reflection. In Meyrowitz, N., editor, *OOPSLA'87 Conference Proceedings*, volume 22(12) of *SIGPLAN Notices*, pages 147–155. ACM Press.

Magee, J. and Kramer, J. (1996). Dynamic Structure in Software Architectures. *ACM Software Engineering Notes*, 21(6):3–14.

Medvidovic, N. and Taylor, R.Ñ. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall.

Milner, R. (1999). *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press.

Shaw, M. (1994). Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *Workshop on Studies of Software Design*.

Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, New Jersey.

Victor, B. and Moller, F. (1994). The Mobility Workbench — A Tool for the $\pi$-Calculus. In Dill, D., editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag.

Wermelinger, M. and Fiadeiro, J. L. (1999). Algebraic Software Architecture Reconfiguration. In *Software Engineering – Proceedings of ESEC/FSE'99*, volume 1687 of *Lecture Notes in Computer Science*, pages 393–409. Springer Verlag.