LEDA: A SPECIFICATION LANGUAGE FOR SOFTWARE ARCHITECTURE *

Carlos Canal Ernesto Pimentel José M. Troya
Depto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga
Campus de Teatinos, 29071 Málaga
Email: {canal,ernesto,troya}@lcc.uma.es

Abstract

Software Architecture refers to the level of design in which a system is described as a collection of interconnected components. Most concepts in the object-oriented paradigm can be applied to Software Architecture, where the more general term *component-oriented* is preferred. However, object-oriented notations often fail to describe the interaction patterns that the components of a system must follow. On the other hand, process algebras are good candidates for the specification of interactive systems, but they are low-level notations, which makes difficult its direct application to the description of complex systems. In this paper we present an Architecture Description Language which combines object-oriented characteristics, such as reusability and inheritance, with the use of process algebras to determine system's compatibility and correctness.

Keywords: Software Architecture, Architecture Description Languages, formal methods, π-calculus, role compatibility, role inheritance.

INTRODUCTION

The increasing complexity of software systems makes evident a lack of notations, methods and tools for dealing with the complexity of programming-in-the-large. In this sense, the term Software Architecture (SA) has been recently adopted, referring to the level of software design in which the system is represented as a collection of computational and data elements interconnected in a certain way [5]. Most concepts coming from the object-oriented paradigm, such as classes and instances, hierarchical composition, inheritance, genericity, polymorphism and dynamic binding can be directly applied to the level of SA. However, at this level the more general term *component-oriented* is preferred, allowing to consider not only objects but architectures, interaction mechanisms and design patterns as first-class concepts of a software architecture [11].

*This work was funded in part by the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) under grant TIC94-0930-C02-01.

Besides, object-oriented specifications of large software systems and, in particular, of distributed and interactive systems often fail to describe the behavioural requirements that a component imposes to those connected to it. Traditionally, interfaces in object-oriented languages are described by the signature of their methods, and there is no explicit description of the interaction patterns, that is, the protocol or sequence of method invocations, that the objects must follow to achieve a correct behaviour of the system.

On the other hand, process algebras are widely accepted for the specification of software systems, in particular for communication protocols and distributed systems. Their formal basis permits the analysis of systems for equivalence, deadlock-freedom and other interesting properties. In this field, we consider that the π-calculus [10], a simple but powerful process algebra, is very well suited for describing complex interactions among components, allowing the analysis of system compatibility and the development of automated verification tools [14]. The π-calculus allows direct expression of *mobility* [4], facilitating the description of dynamic systems, i.e. those whose configuration or *architecture* changes with time.

However, the π-calculus is a low level notation, which makes difficult its direct application to the specification of large systems. Hence, a higher level notation is required.

Our work focuses in the development of formal notations and methods for SA. Our final goal is the development of LEDA, an Architecture Description Language (ADL) which incorporate the concepts of the object-oriented paradigm to obtain hierarchical, modular and reusable specifications. This ADL will use the π-calculus as its formal basis.

From this point of view, the architecture of a software system is described as a collection of components interconnected by several attachments. The interaction patterns that the component follows with respect to each of these attachments are represented by *roles*, specified in π-calculus. As we usually want to attach components whose roles match only partially, equivalence analysis of the roles is not suitable, and we have defined a compatibility relation in the π-calculus which determines the deadlock-freedom of the attachments. This relation is a re-elaboration of the ideas contained in [2], and is summarized here to make this paper self-contained. In order to promote reusability and incremental development, we have also defined a new relation among roles, which we have called inheritance. Role inheritance maintains the derived roles compatible with those which were compatible with its parent. Inheritance can be extended to components, allowing the replacement of a component of a certain system by another one whose roles derive from its parent's.

The structure of this paper is as follows. First, some related works are compared with our proposal. Then, we present LEDA, an ADL based on object-oriented concepts which uses the π-calculus for interface definition and verification. The syntax and main characteristics of the language are shown by meas of an example. Next, relations of role compatibility and role inheritance are defined in the context of the calculus, and their application to LEDA is shown. Some properties of these relations are stated, but due to space limitations, formal proofs of the theorems are omitted. Complete proofs can be found in [3]. Finally, the syntax of the π-calculus is briefly explained in the Appendix.

RELATED WORK

In the last years, several proposals of ADLs have been presented. These proposals describe software architectures from a compositional point of view, and include some form of interface descriptions for components. However, in most of them, like Darwin [8], Rapide [7] or UniCon [13], the analysis of correction of the attachments is reduced to type checking through name equivalence. In [8] the π-calculus is used for defining the semantics of Darwin, where direct expression of mobility in the calculus is used to endow this language with dynamic instantiation mechanisms. In [6] the Chemical Abstract Machine is proposed for architecture specification, allowing the description of dynamic systems, but no higher-level notation is provided.

With respect to compatibility, our proposal follows the ideas developed in [1] for Wright, which uses CSP to determine compatibility of ports and roles. However, the π-calculus allows direct expression of *mobility*, which makes easier the specification of dynamic systems. In fact, as it is stated in [8], CCS or CSP do not seem appropriate for the description of evolving or dynamic structures. Another relevant difference with [1] is that our approach gives a methodology for specifying the roles of a component, and needs no transformation of the roles for compatibility checking.

Besides, none of the proposals found in the literature considers an inheritance relation allowing the addition or redefinition of behaviours. In our proposal this relation preserves relevant properties of the *parent* system, like compatibility.

LEDA: A FIRST APPROACH TO ADLs

In order to determine the suitability of our approach to the specification of software architectures, we are currently designing the language LEDA, an ADL based on the π-calculus.

LEDA permits the specification of software systems in a modular and hierarchical way. A software system is specified in LEDA as the composition of several components. Each of these components is an instance of a component class, and declares an interface formed by several roles. These roles define the interaction patterns that the component follows, indicating the behaviour that it offers and/or requires to its environment, as it will be shown in next section. Additionally to roles, the behaviour of the whole component can be specified, too, and it will serve for prototyping.

Components can be either simple or *composite*. A composite is described as a set of component instances which are interconnected by several attachments to form the composite. The specification of a composite is obtained composing in parallel the specification of its components. The architecture of a system, i.e. the way in which the system is built from its components, is described by the role-to-role attachments between the components of a composite. Each of these attachments are checked for compatibility, which ensures that the system is deadlock-free.

Apart from that, LEDA defines mechanisms to derive new roles from other previously defined, maintaining compatibility. This allows a form of component in-

```
component ProdCons {
  interface empty;
  composition
   p :  Producer;
   c :  Consumer;
   b :  Buffer;
  attachments
   p.Writer(in,wq) <> b.Input(in,wq);
   c.Reader(out,rq) <> b.Output(out,rq);
  }
component Producer {
  interface role Writer(w,q);
  spec ···;
  }
component Consumer {
  interface role Reader(r,q);
  spec ···;
  }
```

```
role Input(i,q) is
   i(item).Input(i,q)
  + q.0;

role Output(o,q) is
   τ.o.(value)ō value.Output(o,q)
  + τ.q̄.0;

component Buffer {
  interface
   role Input(i,iq);
   role Output(o,oq);
  spec ···;
  }
```

Figure 1: Specification of a Producer/Consumer system in LEDA

heritance which is used to perform partial instantiation of software architectures. An architectural pattern, described in LEDA as a composite, can be instantiated replacing part of its components with others that inherit its behavioural properties from the former ones, maintaining in the derived system the architectural properties of the pattern.

The syntax and characteristics of LEDA will be shown using the following example:

Example 1 *A Producer/Consumer system is composed of a Producer, a Consumer and a data storage that we can describe as a Buffer. The Producer generates several items and sends them to the Buffer, ending with a quit event* `wq`*. On the other hand, the Consumer gets each of the items from the Buffer and performs some computations with them. The Consumer process ends when it has got all of the items generated by the Producer, which is notified by the Buffer with an event* `rq`*.*

Notice that this system is dynamic. Not only new processes are created (the items), but also references to these items are sent from the Producer to the Buffer and from that to the Consumer, which uses these references to perform some computations on the items.

The architecture of this system in LEDA is shown in Figure 1, where component ProdCons is a composite which describes the whole system as a composition of a Producer, a Consumer and a Buffer interconnected by two attachments between the roles of these subcomponents. Role specifications can be described outside the components, allowing their reuse in components with similar interfaces. Complete specifications of these components are described in next section.

```
spec Producer(w,q) is
   τ.(it,val)( Item(it,val) | w̄ it.Producer(w,q) )
  + τ.q̄.0

spec Item(item,value) is item̄ value.0;

spec Consumer(r,q) is
   r̄.r(item).item(val).Consumer(r,q)
  + q.0;
```

```
role Writer(w,q) is
  τ.(it)w̄ it.Writer(w,q)
  τ.q̄.0

role Reader(r,q) is
  r̄.r(item).τ.Reader(r,q)
  + q.0;
```

Figure 2: Specification of components `Producer` and `Consumer` with their roles

ROLES AS BEHAVIOURAL INTERFACES OF COMPONENTS

Components can be specified as agents in the π-calculus (see the Appendix for a brief description of the calculus). The links that a component uses for interaction with other components represent its interface. In the π-calculus, this interface is formed by the set of *free names* which appear in the component's agent definition. This interface can be partitioned into several *roles*, each of them referring to the interaction with another component. Thus, a role is an abstraction partially describing the behaviour of a component as seen from another one attached to it. The free names in a role are a subset of those of the corresponding component.

Figure 2 shows the specification of components Producer, and Consumer with the roles that describe how they interact. Since the computations that the Consumer performs with the items it receives are not relevant to its attachment with the Buffer, they are abstracted in role Reader, where they appear simply as τ transitions. The same applies to the Producer; the creation of items is not relevant and is again omitted in role Writer.

The Producer decides locally to engage in writing operations along its link `w`, or to send a quit event $\overline{\mathtt{q}}$, so this behaviour is specified as a local choice in role Writer, combining the sum operator and silent transitions. The rest of the commitments in the roles are globally decided (see the Appendix for an explanation of local and global choice in π-calculus).

The specification of roles as abstractions of components' behaviour is derived from the notion of the *Hiding* operator $(/L)$ defined in [9] for CCS. From this point of view, roles are specified by hiding the free names in a component which are not relevant to the partial interface represented by the role. Any transition of the component indicating communication along the hidden names is replaced in the role by a silent transition. So, a role only contains a subset of the non-silent transitions of the corresponding component and the free names of the role are included in those of the component.

Definition 2 (Role) *A certain process R is a role of a component $Comp$ if*

$$fn(R) \subseteq fn(Comp) \quad and \quad R \equiv Comp/(fn(Comp) - fn(R))$$

(where $\equiv$ denotes bisimilarity).

In the examples above, specifications of components are shown only for clarity, in order to show how the roles are abstractions of the components, and are not necessary for compatibility analysis, as it will be shown. The use of roles instead of whole components for compatibility checking reduces the complexity of the analysis to a great extent.

Role Compatibility

Role compatibility cannot be determined studying the bisimilarity of their specifications, as we usually want to attach processes that match only partially. A formal characterisation of compatibility is given in Definition 4 below.

Roughly, we may say that two roles are compatible if they can engage in at least one transition *(i)*, any local choice of them is supported by the other *(ii,iii)*, and any common transition will lead to compatible processes *(iv)* .

In the definitions below the following shorthand is adopted:

$$\begin{array}{lll} \Longrightarrow & \text{stands for} & (\xrightarrow{\tau})^* \text{ and} \\ \overset{\alpha}{\Longrightarrow} & \text{stands for} & \Longrightarrow\xrightarrow{\alpha} \text{ where } \alpha \neq \tau \end{array}$$

Definition 3 (Related processes) *Two processes, represented respectively by roles P and Q, are related if $fn(P) = fn(Q)$ and*

i. $P \Longrightarrow \mathbf{0} \ \wedge \ Q \Longrightarrow \mathbf{0}$, *or*

ii. $\exists\ \alpha, P', Q'.\ \ P \overset{\alpha}{\Longrightarrow} P' \ \wedge \ Q \overset{\overline{\alpha}}{\Longrightarrow} Q'$

The definition of related processes ensures that, provided they do not progress to inaction, the processes represented by the roles can engage at least in a common transition.

Definition 4 (Relation of compatibility) *A relation $\mathcal{C}$ on processes is a relation of compatibility if $P\ \mathcal{C}\ Q$ implies, for every substitution σ which respects the free names* [1] *in P and Q,*

i. $P\sigma$ *and* $Q\sigma$ *are related,*

ii. if $P\sigma \xrightarrow{\tau} P'$ *then* $P'\ \mathcal{C}\ Q\sigma$,

iii. if $Q\sigma \xrightarrow{\tau} Q'$ *then* $Q'\ \mathcal{C}\ P\sigma$,

iv. if $P\sigma \xrightarrow{\alpha} P' (\alpha \neq \tau)$ *and* $Q\sigma \xrightarrow{\overline{\alpha}} Q'$ *then* $P'\ \mathcal{C}\ Q'$

[1] A substitution respects a set of names if it does not bind any two names in the set [10]. We use distinctions to avoid conflicts among free names in a process. From now on, we will only consider substitutions respecting the free names of the processes involved, even when it is not explicitly mentioned.

Now, we can define the notion of compatible processes as follows.

Definition 5 (Compatible processes) *The processes P and Q are compatible, written $P \rhd Q$, if $P \mathcal{C} Q$ for some compatibility relation $\mathcal{C}$.*

The relation of compatibility above must ensure that no deadlock will arise from interaction in compatible attachments. This will be the aim of Theorems 6 and 7 below.

Proposition 6 *Let P and Q be compatible roles. Then we have that*

$$(P \mid Q)$$

is deadlock-free.

Proof. It can be derived from Definition 5. □

The notion of observability we are interested in, only takes into account silent transitions. Therefore, we consider that a process is deadlock-free when it reaches inaction after a (possibly empty or infinite) sequence of silent transitions.

Theorem 7 *Let Comp be a component with roles $R_1, R_2, \cdots, R_n$. Let P_i be roles such that $P_i \rhd R_i$, for $i = 1...n$. Then we have that*

$$Comp \mid \prod_i P_i$$

is deadlock-free.

Proof. It can be derived from Definition 2 and Proposition 6. □

Compatibility in LEDA

Each of the attachments between components of a composite is checked for compatibility. As a result of Theorem 7 above, this guarantees deadlock-freedom in the composite.

Reconsidering the Producer/Consumer system in Figures 1 and 2, we can derive that role `Writer(in,wq)` $\rhd$ `Input(in,wq)`. They are obviously related and, as `Writer(in,wq)` may commit locally to two τ-transitions, we have to check the first condition of Definition 4 (conditions *(ii),(iii)* and *(iv)* cannot be applied in this case). Condition *(i)* requires that $\overline{\texttt{in}}\,\texttt{it}.\texttt{Writer}(\texttt{in},\texttt{wq}) \rhd \texttt{Input}(\texttt{in},\texttt{wq})$ (1) and $\overline{\texttt{wq}}.\mathbf{0} \rhd \texttt{Input}(\texttt{in},\texttt{wq})$ (2)

(1) They are obviously related. Applying condition *(iv)*, after transitions $\overline{\texttt{in}}\,\texttt{it}$ and $\texttt{in}(\texttt{it})$ respectively, we obtain $\texttt{Writer}(\texttt{in},\texttt{wq}) \rhd \texttt{Input}(\texttt{in},\texttt{wq})$, which is proved by induction.

(2) In this case, we must check that $\mathbf{0} \rhd \mathbf{0}$, which satisfies the conditions of compatibility.

Similarly, we can prove that role `Reader(out,rq)` $\rhd$ `Output(out,rq)`. Therefore, we should be able to ensure that no deadlock will arise from interaction in the attachments of component ProdCons.

COMPONENT INHERITANCE AND DERIVATION

Role Inheritance

A relation of inheritance, similar to that defined in the object-oriented paradigm will be also of use for formal specifications of software components. Inheritance enhances reusability and incremental development, allowing the replacement of modules within the system by specialised versions which maintain some of the properties of the original modules. In order to obtain similar advantages in our context, compatibility must be closed under inheritance. Thus, if two roles Q and R are found compatible, any derived role P related to Q (resp. R) by inheritance should be also compatible with R (resp. P), and we can replace Q (resp. R) by P in any system maintaining compatibility.

We consider role inheritance as a form of strengthening (i.e. greater reliability) of role's behaviour. A derived role must be more predictable, by making fewer local choices, than its parents, while it may also offer new globally-chosen behaviours. On the other hand, derived roles must maintain any global choice appearing in their parents, in order to maintain compatibility. Thus, we can replace a component by another one whose roles are related to the previous ones by inheritance maintaining system's properties of compatibility.

Definition 8 (Role Inheritance) *Let P and Q be roles. P $inherits from$ Q iff for every σ respecting their free names,*

i. *if* $Q\sigma \xrightarrow{\tau} Q'$ *then* $\exists P'.\ P\sigma \xrightarrow{\tau} P'$ *and* Q' $inherits from$ P'

ii. *if* $P\sigma \xrightarrow{\alpha} P' (\alpha \neq \tau)$ *then* $\exists Q'.\ Q\sigma \xrightarrow{\alpha} Q'$ *and* Q' $inherits from$ P'

iii. *if* $P\sigma \stackrel{\alpha}{\Longrightarrow} P' (\alpha \neq \tau)$ *and* $subject(\alpha) \in fn(Q)$
then $\exists Q'.\ Q\sigma \stackrel{\alpha}{\Longrightarrow} Q'$ *and* Q' $inherits from$ P'

iv. *if* $P\sigma \xrightarrow{\tau} P'$ *then* $\exists P_o, Q_o.\ P\sigma \xrightarrow{\tau} P_o$ *and* $Q\sigma \xrightarrow{\tau} Q_o$ *and* Q_o $inherits from$ P_o

The following result shows that role inheritance preserves compatibility.

Theorem 9 *Let Q and R be roles. Let $Q \rhd R$ and P $inherits from$ Q. Then we have that*

$$((fn(P) - fn(Q))P \rhd R$$

Proof. It can be derived from Definitions 5 and 8. □

```
component MyProducer inherits Producer {
  interface role MyWriter(w,q)
   redefines Writer(w,q) is (it1,it2,it3)
     (w̄ it1. w̄ it2. w̄ it3. q̄.0);

  spec MyProducer(w,q) is (it1,it2,it3)
   ( Item(it1,BLACK) |
     Item(it2,WHITE) |
     Item(it3,BLACK) |
     w̄ it1. w̄ it2. w̄ it3. q̄.0);
  }

component MyConsumer inherits Consumer {
  spec MyConsumer(r,q) is
     r̄.r(item).item(colour).
       ([colour=BLACK](b)(Black | MyConsumer(r,q))
  + q.0;

  spec Black is τ.0;
  }
```

Figure 3: Specification of components `MyProducer` and `MyConsumer`

Component inheritance in LEDA

The relation of role inheritance in Definition 8 permits the definition of inheritance among components and also system's derivation maintaining compatibility.

Definition 10 (Component Inheritance) *Let $Comp_P$, $Comp_Q$ be components, with roles $\{P_i\}_i$ and $\{Q_j\}_j$ respectively.*

$Comp_P$ inherits from $Comp_Q$ iff for each role $P \in \{P_i\}_i, \exists Q \in \{Q_j\}_j$, where P inherits from Q.

As role inheritance is reflexive, a component inherits from another simply by redefining some of its roles. The rest of the parent's roles are inherited as they are.

Role inheritance is expressed in LEDA in two different ways. First, a child role may extend its parent's behaviour, adding some new global choices to its parent's specification. Second, a child role may redefine its parent, restricting local choices as indicated in Definition 8.

Figures 3 and 4 show components which inherit from those described in Figures 1 and 2. Component `MyProducer` generates three coloured items, sends them by the link `w` and ends with a `q` event. We can check that role `MyWriter` inherits from `Writer`, and therefore component `MyProducer` inherits from `Producer`. To verify this relation of inheritance, we apply conditions *(iii)* and *(iv)* from Definition 8.

(iii) Since $\texttt{MyWriter}(\texttt{w},\texttt{q}) \xrightarrow{\overline{\texttt{w}}} \overline{\texttt{w}}\ it2.\ \overline{\texttt{w}}\ it3.\ \overline{\texttt{q}}.\ \mathbf{0}$ and $\texttt{Writer}(\texttt{w},\texttt{q}) \stackrel{\overline{\texttt{w}}}{\Longrightarrow} \texttt{Writer}(\texttt{w},\texttt{q})$,

applying twice again the same transition, we get that $\texttt{MyWriter}(\texttt{w},\texttt{q})$ inherits from $\texttt{Writer}(\texttt{w},\texttt{q})$ if $\overline{\texttt{q}}.\mathbf{0}$ inherits from $\mathbf{0}$, and by condition *(iii)* again, we arrive at $\mathbf{0}$

```
role OStack(o,isempty,oq)
  redefines Output(o,oq) adding
   isempty empty.Ostack(o,isempty,oq);

component Stack inherits Buffer {
  interface role OStack(o,isempty,oq);

  spec Stack(i,o,iq,oq) is
   (node)Stack(i,o,iq,oq,node,TRUE,TRUE);

  spec Stack(i,o,iq,oq,node,empty,writer) is
   i(item).(new)( Node(new,item,node,empty) |
     Stack(i,o,iq,oq,new,FALSE,writer) )
  + iq.Stack(i,o,iq,oq,node,empty,FALSE)
  + [empty=FALSE]o.node(item,next,empty).ō item.
     Stack(i,o,iq,oq,next,empty,writer)
  + [empty=TRUE][writer=FALSE]ōq.0
  + isempty empty.Stack(i,o,isempty,iq,oq,node,empty,writer);

  spec Node(node,item,next,last) is node item next last.0;
  }
```

Figure 4: Specification of component `Stack`

inherits from $\mathbf{0}$, which satisfies the conditions of inheritance.

(iv) $\texttt{Writer}(\texttt{w},\texttt{q})$ has two τ-transitions. In particular for $\texttt{Writer}(\texttt{w},\texttt{q}) \xrightarrow{\tau} \overline{\texttt{q}}.\mathbf{0}$, we have that $\overline{\texttt{q}}.\mathbf{0}$ inherits from $\texttt{Writer}(\texttt{w},\texttt{q})$ if $\overline{\texttt{q}}.\mathbf{0}$ inherits from $\overline{\texttt{q}}.\mathbf{0}$, which satisfies conditions *(ii)* and *(iii)*.

On the other hand, component `MyConsumer` gets items from its input link `r` and performs some computations with those which are black. As these computations are internal, its interface is not affected by them. Thus `MyConsumer` inherits from `Consumer` without redefining its role. Finally, Figure 4 shows an implementation of component `Buffer` which takes the form of a stack. Apart from storage and retrieval of data items, the stack can inform whether it is empty or not. This new service, not considered in Buffer's roles, is added to role `OStack`, which results a heir of `Output`, as can be checked using Definition 8. Thus, component `Stack` inherits from `Buffer` (in fact, it *implements* the buffer, since we hadn't provided a specification for the `Buffer` yet).

This relations of inheritance among components can be checked in LEDA, and they ensure that we could built a Producer/Consumer system using the components in Figures 3 and 4. However, it would be of little use if we had to write again the system using the new components. A mechanism of component instantiation, similar to the use of generic classes in object-oriented languages, is needed. We have called this mechanism in LEDA *component derivation*.

Component derivation in LEDA

In order to promote reusability of software architecture specifications, LEDA offers the possibility to replace any component in a composite by another one which inherits from it. In our example, this replacement is performed as follows:

```
myProdCons :  ProdCons(p:MyProducer, b:Stack, c:MyConsumer);
```

This example shows how incremental specification of components, being careful of fulfilling the requirements of role inheritance, preserves deadlock-freedom when replacing the original components by the new ones. This property serves for the same purposes as data polymorphism in the object-oriented paradigm.

Thus, we can consider a component class in LEDA as generic class, in which any of its subcomponents may act as a generic parameter for component classes. These parameters can be instantiated to obtain new components which maintain the architectural properties (including deadlock-freedom in the attachments) of the original component.

CONCLUSIONS

ADLs address the description of software systems during the design process. In this paper we have presented LEDA, an ADL which combines object-orientation concepts with the use of the π-calculus, a process algebra very well suited for formal specification of dynamic systems.

Using LEDA, the specification of roles as interfaces of components is enough in order to verify system's composability from the compatibility of each role-to-role attachment. Besides, the specification of system's components in π-calculus serves as a first prototype of the system, which can be checked against user's requirements.

We have developed a relation of compatibility for the π-calculus which guarantees that the attachments of the roles are deadlock-free. Incremental specification and reusability are promoted by the definition of a relation of inheritance among roles specified in the calculus. This relation of inheritance is extended to components, and permits to consider any component specification in LEDA as a generic pattern which may be instantiated in order to obtain particular systems which maintain the architectural properties of the pattern.

References

[1] R. Allen and D. Garlan. "Formal Connectors". *Technical Report* CMU-CS-94-115. Carnegie-Mellon, March 1994. Available at: <ftp://reports.adm.cs.cmu.edu>

[2] C. Canal, E. Pimentel and J.M. Troya. "Software Architecture Specification with π-calculus". *Jornadas de Trabajo en Ingenieria del Software*, (Sevilla, Nov. 1996): 31–40.

[3] C. Canal, E. Pimentel and J.M. Troya. "A Formal Definition of Compatibility and Inheritance for Software Architectures". *Technical Report*. University of Málaga, 1997.

[4] U. Engberg and M. Nielsen. "A Calculus of Communicating Systems with Label-passing". *Technical Report DAIMI PB-208*, Computer Science Dept., University of Aarhus, 1986.

[5] D. Garlan and D. E. Perry. "Introduction to the Special Issue on Software Architecture". *IEEE Transactions on Software Engineering*, vol. 21, no. 4 (April 1995): 269–274.

[6] P. Inverardi and A. L. Wolf. "Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model". *IEEE Transactions on Software Engineering*, vol. 21, no. 4 (April 1995): 373–386.

[7] D. C. Luckham et al. "Specification and Analysis of System Architecture using Rapide". *IEEE Transactions on Software Engineering*, vol. 21, no. 4 (April 1995): 336–355.

[8] J. Magee and J. Kramer. "Dynamic Structure in Software Architectures". *Proc. of ACM SIGSOFT'96 Symposium on the Foundations of Software Engineering* (San Francisco, Oct. 1996): 3–14.

[9] R. Milner. *Communication and Concurrency*, Prentice Hall, 1989.

[10] R. Milner, J. Parrow and D. Walker. "A Calculus of Mobile Processes, Parts I and II". *Journal of Information and Computation*, vol. 100, (1992): 1–77.

[11] O. Nierstrasz. "Requirements for a Composition Language". *Proc. of ECOOP'94*, LNCS 924, Springer Verlag, 1995: 147–161.

[12] D. Sangiorgi. "A theory of bisimulation for the π-calculus". *Technical Report* ECS-LFCS-93-270, University of Edinburgh, June 1993. Available at:
<http://www.dcs.ed.ac.uk/publications/lfcsreps>

[13] M. Shaw et al. "Abstractions for Software Architecture and Tools to Support Them". *IEEE Transactions on Software Engineering*, vol. 21, no. 4 (April 1995): 314–335.

[14] B. Victor. "A Verification Tool for the Polyadic π-calculus". Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994.

APPENDIX: THE π-CALCULUS

The π-calculus is a process algebra derived from CCS. It is specially suited for the description and analysis of concurrent systems. A concurrent system is specified in the π-calculus as a collection of processes or *agents* which interact by means of links or *names* (we denote by $\mathcal{N}$ the set of names). These names can be considered as bidirectional channels that may be shared by several processes. The restriction of the scope of a name allows to establish links that are private to a group of processes.

The calculus is very simple but expressive enough. Although only names can be transmitted along links, name transmission, together with agent instantiation, allows the reconfiguration of the system.

Processes are built from names and operators by following the following syntax:

$$P ::= \mathbf{0} \mid \alpha.P, \alpha \in \mathcal{A} \mid P_1 + P_2 \mid P_1|P_2 \mid (x)P,\ x \in \mathcal{N} \mid [x = z]P \mid A(\tilde{x})$$

where $\tilde{x}$ is a sequence of elements in $\mathcal{N}$, $A(\tilde{x})$ is an agent definition equation, and the set of atomic actions is given by:

$$\mathcal{A} = \{x(y), \bar{x}y :\ x, y \in \mathcal{N}\} \cup \{\tau\}$$

The process $\mathbf{0}$ represents the inactive process. The restriction operator $(x)P$ hides the name x, restricting its scope to P. A process with a prefix action, such as $\alpha.P$ performs an action α, and then behaves like P. Silent transitions (given by τ) model internal actions. An action $\bar{x}y$ is called *negative prefix*, and it represents the sending of the name y along the channel x. Similarly, an action $x(y)$ is called *positive prefix*, and corresponds to the complementary action. A summation of two processes, $P_1 + P_2$, behaves like P_1 or P_2 (non-deterministically). The parallel composition of two processes, $P_1|P_2$, may evolve independently or synchronise, matching a positive prefix in P_1 (resp. P_2) with the corresponding negative prefix in P_2 (resp. P_1) as indicated by the following reduction rule:

$$(\cdots + \bar{x}z.P_1 + \cdots) \mid (\cdots + x(y).P_2 + \cdots) \xrightarrow{\tau} P_1|P_2\{z/y\}$$

This synchronisation will be externally observed as a silent transition τ. Since both process agree in the commitment to complementary transitions, this synchronisation is called a *global choice*. On the other hand, *local choices* are expressed in the calculus by combining the summation operator with silent actions. Hence, a process like

$$(\cdots + \tau.\bar{x}w.P_1 + \tau.y(z).P_2 + \cdots)$$

may decide by itself the commitment to $\bar{x}w.P_1$ or $y(z).P_2$, with independence of its context.

Finally, the *free names* of a process P, $fn(P)$, are those names occurring in P not bound by a positive prefix or by a restriction.

For a detailed description of π-calculus we refer to [10]. The transition system that we are considering can be found in [12].