LEDA: An Object-Oriented Approach to Software Architecture Specification *

Carlos Canal Ernesto Pimentel José M. Troya

Depto. de Lenguajes y Ciencias de la Computación, University of Málaga

Campus de Teatinos, 29071 Málaga, Spain

{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. On the other hand, process algebras are good candidates for system specification. However, they are low-level notations, which makes difficult its direct application to the specification 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.

Keywords: meta architectures, formal calculus, concurrency, Software Architecture, compatibility

1 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 [3]. Although object-orientation can be applied to different levels of abstraction, in SA the more general term *component* is preferred [9], allowing to consider not only objects but architectures, interaction mechanisms and design patterns as first-class concepts of a software architecture. Most object-oriented concepts, such as classes and instances, hierarchical composition, inheritance, polymorphism and dynamic binding can be directly applied to the so called component-oriented paradigm.

However, 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.

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

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 permit the analysis of systems for equivalence, deadlock-freedom and other interesting properties. In this field, we consider that the π-calculus [8], 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 [11]. The π-calculus allows direct expression of mobility, 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. In order to promote reusability and incremental development, we have also defined a relation of inheritance among roles which maintains the derived roles compatible with those which were compatible with its parent. Role 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. Next, relations of role compatibility and role inheritance are defined in the context of the calculus, and some properties of these relations are stated. Due to space reasons, formal proofs of these theorems are omitted, but they can be found in [2]. Finally, the syntax of the π-calculus is briefly explained in Appendix A.

2 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 [6], Rapide [5] or UniCon [10], the analysis of correction of the attachments is reduced to type checking through name equivalence. In [6] 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 [4] 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 [6], 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.

3 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.

A software system is specified in LEDA as the composition of several components (Figure 1). 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 (as the `Consumer`, `Producer` and `Stack`) or composite, as the whole system in Figure 1, left. A composite is described as a set of component instances. The specification of a composite is obtained composing in parallel the specification of its components. The architecture of the systems, i.e. the way in which the system is built from its components, is described by the role-to-role attachments between components.

At its current state, LEDA looks like most ADL proposals (see related work). However, the definition of role compatibility and inheritance, combined with the possibility of dynamic binding and lazy instantiation which derive from the dynamic characteristics of the π-calculus, determine the originality of our approach.

```
component ProducerConsumerSystem {
  interface none;
  composition
   p :  Producer;
   c :  Consumer;
   s :  Stack;
  attachments
   p.RoleProducer <> s.Writer;
   c.RoleConsumer <> s.Reader;
  }
```

```
component Producer {
  interface RoleProducer;
  spec Producer;
}
component Consumer {
  interface RoleConsumer;
  spec Consumer;
}
component Stack {
  interface Reader, Writer;
  spec Stack;}
```

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

4 Roles as behavioural interfaces of components

Components can be specified as agents in the π-calculus (see Appendix A 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. So, 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.

Figures 2, 3 show the specification of components `Producer`, `Consumer` and `Stack` with the roles that describe how they interact. As the computations that the `Consumer` performs with the items received are not relevant to its attachment with the `Stack`, 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 `Stack` decides locally to engage in reading operations along its link `out`, or to send a `rquit` event, so this behaviour is specified as a local choice in role `Reader`, combining the sum operator and silent transitions. The rest of the commitments in the system are globally decided (see Appendix A 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 [7] for CCS. From this point of view, roles are specified by means of hiding the free names in a component which are not relevant to the 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 4.1 (Role) *A certain process R is a role of a component $Comp$ if*

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

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

```
Producer(in,wquit) = (item1,item2,item3)
   ( Item(item1,BLACK) |
   Item(item2,WHITE) |
   Item(item3,BLACK) |
   in item1. in item2. in item3. wquit.0)

Item(item,color) = item color.0

RoleProducer(in,wquit) = (item)
   (in item. in item. in item. wquit.0)
```

```
Consumer(out,rquit,count) =
  out.out(item).item(color).
  ([color=BLACK]count.0
  | Consumer(out,rquit,count))
  + rquit.0

RoleConsumer(out,rquit) =
  out.out(item).τ.RoleConsumer(out,rquit)
  + rquit.τ.0
```

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

```
Stack(in,out,wquit,rquit,node,empty,writer) =
  in(item).(new)( Node(new,item,node,empty) |
   Stack(in,out,wquit,rquit,new,FALSE,writer) )
  + wquit.Stack(in,out,wquit,rquit,node,empty,FALSE)
  +[empty=FALSE] out.node(item,next,empty).out item.
   Stack(in,out,wquit,rquit,next,empty,writer)
  + [empty=TRUE][writer=FALSE]rquit.0

Node(node,item,next,last) = node item next last.0
```

```
Writer(in,wquit) =
  in(item).Writer(in,wquit) +
  wquit.0

Reader(out,rquit) =
  τ.out.(value)out value.
   Reader(out,rquit) +
  τ.rquit.0
```

Figure 3: Component `Stack` with its roles

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.

5 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 5.2 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:

$$\Longrightarrow \quad \text{stands for} \quad (\xrightarrow{\tau})^* \quad \text{and}$$
$$\stackrel{\alpha}{\Longrightarrow} \quad \text{stands for} \quad \Longrightarrow\xrightarrow{\alpha} \quad \text{where } \alpha \neq \tau$$

Definition 5.1 (Related processes) *Two processes, represented respectively by roles P and R, are related if $fn(P) = fn(R)$ and*

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

ii. $\exists\ \alpha, P', R'.\ \ P \stackrel{\alpha}{\Longrightarrow} P' \ \wedge \ R \stackrel{\overline{\alpha}}{\Longrightarrow} R'$

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

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

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

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

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

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

Definition 5.3 (Compatible processes) *The processes P and R are compatible, written $P \rhd R$, if $P\ \mathcal{C}\ R$ for some compatibility relation $\mathcal{C}$.*

Reconsidering the Producer/Consumer system in Figures 2 and 3, we can derive that `RoleProducer` $\rhd$ `Writer`. They are obviously related, and only the fourth condition has to be checked (conditions *(ii)* and *(iii)* cannot be applied in this case). Since

$\texttt{RoleProducer} \xrightarrow{\overline{\texttt{in}}\, w} \overline{\texttt{in}}\ w.\ \overline{\texttt{in}}\ w.\ \overline{\texttt{wquit}}.\ \mathbf{0}$ and $\texttt{Writer} \xrightarrow{in(w)} \texttt{Writer}$,

applying twice the same pair of transitions, we obtain respectively $\overline{\texttt{wquit}}.\ \mathbf{0}$ and `Writer` which are found compatible checking again condition *(iv)*. Similarly, we can prove that `RoleConsumer` $\rhd$ `Reader`. Therefore, we should be able to ensure that no deadlock will arise from interaction in a concrete attachment. This will be the aim of Theorems 5.4 and 5.5 below.

Proposition 5.4 *Let P and R be compatible roles. Then, $P|R$ is deadlock-free.*

Proof. It can be derived from Definition 5.3. □

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 5.5 *Let Comp be a connector 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*

$$Comp|\prod_i P_i$$

is deadlock-free.

Proof. It can be derived from Definition 4.1 and Proposition 5.4. □

6 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 P and R are found compatible, any derived role P' related to, P (resp. R) by inheritance should be also compatible with R, (resp. P) and we can replace P by P' in any system maintaining compatibility.

We consider role inheritance as a form of strengthening (i.e. greater reliability) of role's behaviour. So, derived roles must be more predictable, by making fewer local choices,

```
component Stack1 inherits Stack {
  interface
   Reader1 inherits Reader,
   Writer1 inherits Writer;
}
```

```
Reader1(out,rq,isempty) =
    Reader(out,rq)
  + isempty(empty).
    Reader1(out,rq,isempty)

Writer1(in,wq,isempty) =
    Writer(in,wq)
  + isempty(empty).
    Writer1(in,rq,isempty)
```

Figure 4: A new version of the `Stack` with the derived roles

and they may also offer new globally-chosen behaviours. On the other hand, they must maintain any global choice present in their parents, in order to maintain compatibility. Thus, we can replace a component by another whose roles are related to the previous ones by inheritance maintaining system's properties of compatibility.

Definition 6.1 (Role Inheritance) *Let P and Q be roles. P inherits Q iff for every σ respecting their free names,*

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

ii. if $Q\sigma \xrightarrow{\alpha} Q'$ then $\exists P'.\ P\sigma \xrightarrow{\alpha} P'$ and P' inherits Q'

iii. if $P\sigma \xrightarrow{\alpha} P'$ and $Q\sigma \stackrel{\alpha}{\Longrightarrow} Q'$ then P' inherits Q'

iv. if $Q\sigma \xrightarrow{\tau} Q'$ then $\exists Q''.\ Q\sigma \xrightarrow{\tau} Q''$,
where $P\sigma$ inherits Q'' or $P\sigma \xrightarrow{\tau} P'$ and P' inherits Q''

The following result shows that role inheritance preserves compatibility.

Theorem 6.2 *Let P and R be roles. Let $P \rhd R$. Let P' inherits P. Then we have that $P' \rhd R$.*

Proof. It can be derived from Definitions 5.3 and 6.1. □

Figure 4 shows a new version of the `Stack` component, in which a new service, which indicates whether the stack is empty or not, is added. The new roles inherit from those defined in Figure 3. Therefore we can replace the former stack by this derived version maintaining the compatibility in system's attachments. 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.

7 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 the 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.

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. "A Formal Definition of Compatibility and Inheritance for Software Architectures". *Technical Report*. University of Málaga, 1997.

[3] 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, pp. 269–274.

[4] 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, pp. 373–386.

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

[6] J. Magee and J. Kramer. "Dynamic Structure in Software Architectures". *Proc. of ACM SIGSOFT'96 Symposium on the Foundations of Software Engineering*, San Francisco, October 1996, pp. 3-14.

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

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

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

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

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

A 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 enough expressive. 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 indicates 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. As 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 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, $\mathrm{fn}(P)$, 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 [8].