A Formal Specification Language for the Description of Architectural Patterns in Software Systems

Carlos Canal **Ernesto Pimentel** **José M. Troya**

Depto. de Lenguajes y Ciencias de la Computación

Universidad de Málaga

Campus de Teatinos, 29071 Málaga, España

{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 their direct application to the description of complex systems. In this paper we present an Architecture Description Language which combines object-oriented characteristics, such as inheritance and pattern instantiation, with the use of process algebras to determine system compatibility and well-formedness. The language permits the description of architectural patterns which can be instanced to obtain specific architectures for different systems.

Keywords: Software Architecture, Architecture Description Languages, Architecture Patterns, π-calculus, compatibility, inheritance.

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 [Garlan 95]. Most concepts coming from the object-oriented paradigm, such as classes and instances, hierarchical composition, inheritance, parameter-ization, 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 [Nierstrasz 94].

Object-oriented specifications of distributed and interactive systems often fail to describe the behavioral 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 protocol or sequence of method invocations, that the objects must follow to achieve a correct behavior 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 [Milner 92], a simple but powerful process algebra, is very well suited for describing complex interactions among components, allowing formal analysis of the specifications, and the development of automated verification tools [Victor 94]. The π-calculus allows direct expression of *mobility* [Engberg 86], facilitating the description of dynamic systems, i.e. those whose configuration or *architecture* changes with time. However, the π-

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

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 incorporates 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 ensures deadlock freedom in the attachments. In order to promote reusability and incremental development, we have also defined a relation of derivation among roles, which we have called inheritance. Role inheritance maintains the derived roles compatible with those which were compatible with their parents. A formal definition of these relations is out of the scope of this paper, but it can be found in [Canal 96], where the most relevant properties of role compatibility and inheritance are stated. Nevertheless, the relations are briefly presented here to make this paper self-contained. A more detailed description, including formal proofs of all their properties can be found in [Canal 97].

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. LEDA incorporates a mechanism of architecture instantiation by component replacement. Thus, a software architecture specified in LEDA can be considered as a *framework* in the sense of [Pree 95]. Component frameworks represent the highest level of reusability in Software development: not only source code and single components, but also architectural design is reused in applications built on top of the framework [Pree 96]. LEDA frameworks can be partially instanced and reused as many times as needed. As LEDA specifications can be verified for compatibility and well-formedness, this promotes both software reusability and quality.

The structure of this paper is as follows. First, some related works are compared with our proposal. Then, we give a short introduction to the π-calculus. Next, we present LEDA, an ADL based on object-oriented concepts which uses the π-calculus for interface definition and verification. The main characteristics of the language are shown by means an example. Next, relations of role compatibility and role inheritance are defined in the context of the π-calculus, and their application to LEDA is shown.

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 [Magee 96] or UniCon [Shaw 95] , the analysis of correction of the attachments is reduced to type checking through name equivalence. In [Magee 96] 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 [Inverardi 95] the Chemical Abstract Machine is proposed for the description of dynamic systems, but no higher-level notation is provided.

With respect to compatibility, our proposal follows the ideas developed in [Allen 94] 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.

None of the proposals found in the literature considers an inheritance relation allowing the addition or redefinition of behaviors. In our proposal this relation preserves relevant properties of the *parent* system, like compatibility, and is extended to components, which permits the use of LEDA to describe software frameworks.

3. The π-calculus

The π-calculus is a process algebra derived from CCS. A concurrent system is specified in the π-calculus as a collection of processes or *agents* which interact by means of links or *names*. These names can be considered as bidirectional channels that may be shared by several processes. 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:

$$P ::= \mathbf{0} \mid \alpha.P \mid P_1+P_2 \mid P_1|P_2 \mid (x)P \mid [x=z]P \mid A(x^*)$$

where N is the set of names, x, z ∈ N, x* ∈ N*, A(x*) is an agent definition equation, and A is the set of atomic actions, given by:

$$A = \{x(y),\ \hat{}x\langle y\rangle : x,y \in N \cup \{\tau\}$$

The process **0** represents the inactive process. The restriction operator (x)P hides the name x, restricting its scope to P. This operator allows to establish links that are private to a group of processes. 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 ^x<y> is called *negative prefix*[2] , 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. 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. A summation of two processes, P_1+P_2, behaves like P_1 or P_2. The parallel composition of two processes, $P_1|P_2$, may evolve independently or synchronize, 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:

$$(\ldots + \hat{}x\langle z\rangle.P_1 + \ldots) \mid (\ldots + x(y).P_2 + \ldots) \xrightarrow{\tau} P_1|P_2\{z/y\}$$

This synchronization will be observed as a silent transition τ. As both process agree in the commitment to complementary transitions, this synchronization is called a *global choice*, and will be used to model deterministic behavior. On the other hand, *local choices* model nondeterministic behavior, and are expressed in the calculus combining the summation operator with silent actions. Hence, a process like

$$(\ldots + \tau.\hat{}x\langle w\rangle.P_1 + \tau.y(z).P_2 + \ldots)$$

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

For a detailed description of π-calculus we refer to [Milner 92]. The transition system that we are considering can be found in [Sangiorgi 93].

---

[2] The actual notation for negative prefixes is $\bar{x}y$. However, ^x<y> is usually found in textual environments.

4. LEDA: A First Approach to ADLs

LEDA permits the specification of software systems in a modular and hierarchical way. The syntax of language LEDA is shown in Figure 1. A software architecture 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,

<*LEDA architecture*> ::= [<*role definition list*>] { <*component*> }

<*component*> ::= **component** <*component name*> '{'
interface [**empty** | <*role definition list*>]
[**composition** <*instance declaration list*>
attachments <*attachments list*>]
[<*specification list*>]
'}'

<*role definition list*> ::= { **role** <*role definition*> ';' }

<*role definition*> ::= <*role definition equation*> [**is** <π-*calculus specification*>] ';'

<*instance declaration list*> ::= { <*instance name list*> ':' <*component name*>
['(' <*instance declaration list*> ')'] ';' }

<*instance name list*> ::= { <*instance name*> ',' }

<*attachments list*> ::= { <*attachment*>';' }

<*attachment*> ::= <*instance name*> '.' <*role definition equation*> '<>'
<*instance name*> '.' <*role definition equation*> ';'

<*specification list*> ::= { <*specification*> }

<*specification*> ::= **spec** <*agent definition equation*> **is** <π-*calculus specification*> ';'

Figure 1. Syntax of LEDA

```
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.(v)^o v.Output(o,q)
 + τ.^q.0;

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

Figure 2. Specification of a Producer/Consumer system in LEDA.

indicating the behavior that it offers and/or requires to its environment, as it will be shown in next section. Additionally to roles, the behavior 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 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.

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

Example 4.1. *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, using an operation* `in`*, and it ends sending a quit event* `wq`*. On the other hand, the consumer gets each of the items from the buffer, invoking operation* `out` *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 the latter 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 2, 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 and their roles are described in next section.

5. Roles as behavioral interfaces of components

Components can be specified as agents in the π-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 behavior 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.

```
spec Producer(w,q) is
   τ.(it,v)( Item(it,v) |
             ^w<it>.Producer(w,q) )
 + τ.^q.0

spec Item(item,value) is ^item<val>.0;

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

role Writer(w,q) is
   τ.(it)^w<it>.Writer(w,q)
 + τ.^q.0

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

Figure 3. Specification of components `Producer` and `Consumer` with their roles

Figure 3 shows the specification of components Producer, and Consumer with the roles that describe how they interact. Since these components are simple and they have a single role, specifications of components and roles are more or less the same. However, the specification of more complex components which have several roles, as the `Stack` that will be shown in Figure 5, will be greatly simplified in its roles.

The `Producer` in Figure 3 may decide (nondeterministically) to create a new item (with `Item(it,val)`), and send it to the buffer with `^w<it>`, or it may quit sending an event `^q`. Nondeterminism is specified in π-calculus by local choices, using silent actions τ before the corresponding transitions. Role `Writer` specifies the interface of the `Producer`, as it is seen from its environment. Basically, internal actions, like the creation of the items, are omitted. On the other hand, the `Consumer` may perform two transitions: it may send an event `^r`, to initiate a reading operation, or it may receive an event q, which indicates that the `Buffer` is empty and also that the `Producer` has quitted the system, and therefore the `Consumer` must quit, too. Since the commitment to any of these transitions is not decided by the `Consumer`, but it depends on internal conditions of the `Buffer`, the are expressed in the calculus as global choices (not using silent transitions). Again, the role `Reader` implements the interface of the `Consumer`, hiding the internal actions that the latter performs with the items received.

The notion of role as an abstraction of a component's behavior, derives from the *Hiding* operator (/L) defined in [Milner 89] 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.

In the examples above, specifications of components are shown only for clarity, in order to show how the roles are abstractions of them, but they 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.

5.1 Role Compatibility

Role compatibility cannot be determined studying the equivalence of their specifications, as we usually want to attach processes that match only partially. A formal characterization of compatibility is given in [Canal 96].

Roughly, we may say that two roles P and Q are compatible, written $P \lozenge Q$, if *(i)* they can engage in at least one common transition, avoiding to consider compatible processes not related to each other (i.e. with disjoint sets of free names) which would deadlock when composed in parallel; *(ii)* any local choice in one of them is supported by the other; and *(iii)* any common transition leads to compatible processes.

The relation of compatibility among roles ensures that no deadlock will arise from interaction in compatible attachments. This is the aim of Proposition 5.1 and Theorem 5.2 below.

Proposition 5.1. *Let P and Q be compatible roles. Then we have that*

$$P \mid Q$$

is deadlock-free.

Thus, the attachment of two compatible roles, represented here by their parallel composition, is deadlock-free. 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 5.2 *Let Comp be a component with roles R_1, R_2,..., R_n. Let P_i be roles such that $P_i \lozenge R_i$, for i=1...n. Then we have that*

$$Comp \mid \prod_{i} P_i$$

is deadlock-free.

This theorem is a generalization of the preceding one, and ensures that the attachment of a component to others whose roles are compatible with those of the former is deadlock-free. Therefore, if every attachment in the architecture of a certain system is found compatible, no deadlock will arise from the interaction among components, and the system can be safely built.

5.2 Compatibility in LEDA

Each of the attachments between components of a composite is checked for compatibility. As a result of

Theorem 5.2 above, this guarantees deadlock freedom in the composite. Reconsidering the Producer-Consumer in Figure 2 and Figure 3, we can derive that `Writer(in,wq)` ◊ `Input(in,wq)`. Their free names are identical, so they are related and can interact through their common links. Since `Writer(in,wq)` may decide to put a new item on the buffer (by committing locally to the transition that sends it along link `in`) or to quit (by sending en `wq`), we have to check that the buffer, represented here by the role `Input(in,wq)`, is able to accept these decisions. Notice that, as the complementary actions in `Input(in,wq)` are globally decided, the buffer is able to follow any of these decisions, and the two roles are compatible. Similarly, we can prove that role `Reader(out,rq)` ◊ `Output(out,rq)`. Therefore, we should be able to ensure that no deadlock will arise from interaction in the attachments of component ProdCons.

6. Inheritance and Derivation

6.1 Role Inheritance

In the object-oriented paradigm inheritance refers to a relation among object classes by which a heir class inherits the properties (methods and attributes) of its parent classes, while adding its own properties. The inherited properties may be redefined, usually under certain restrictions, but roughly speaking, we may say that the interface of the heir class includes those of its parents. A relation of inheritance would be also of use for specifications of software components.

However, in our context, the interface of a component is defined not only by the signature of its properties (i. e. the signature of its roles), but this interface also includes the behavioral patterns described in the roles. Thus, a child component inherits the roles of its parents, while role redefinition is restricted by several conditions, which ensure that compatibility is maintained.

These conditions define what we have called a relation of inheritance among roles and ensures that role compatibility is 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.

Once again, a formal definition of role inheritance for the π-calculus can be found in [Canal 96]. We consider role inheritance as a form of strengthening (i.e. giving greater reliability) of role's behavior. So, derived roles must be more predictable, by making fewer local choices, and they may also offer new globally-chosen behaviors. On the other hand, derived roles must present any global choice appearing in their parents, in order to maintain compatibility.

The following result ensures that role inheritance preserves compatibility.

Theorem 6.1. *Let P, Q and R be roles. Let* $Q \lozenge R$ *and P inherits from Q. Then we have that*

$$P \lozenge R$$

Thus, we can replace a component by another one whose roles are related to the previous ones by inheritance, maintaining system properties of compatibility. Role inheritance define the restrictions which allow polymorphism of behavior, and promotes both incremental specification and reusability.

6.2 Component inheritance in LEDA

The relation of role inheritance permits the definition of inheritance among components and also system derivation maintaining compatibility.

Definition 6.2. (Component inheritance) *Let* $Comp_P$ *and* $Comp_Q$ *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.*

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

  spec MyProducer(w,q) is (it1,it2,it3)
    ( Item(it1,BLACK) |
      Item(it2,WHITE) |
      Item(it3,BLACK) |
      ^w<it1>.^w<it2>.^w<it3>.^q.0 );
  }

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

  spec Black is τ.0;
  }
```

Figure 4. Specification of components `MyProducer` and `MyConsumer`

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 can be expressed in LEDA in two different ways. First, a child role may extend its parent's behavior, adding some new globally-decided choices to its parent's specification. Second, a child role may redefine its parent, restricting its local choices but maintaining compatibility with the roles which were compatible with its parent.

Figure 4 and Figure 5 show components which inherit from those described in Figure 2 and Figure 3. Component `MyProducer` generates three colored items, sends them by the link `w` and ends with a `q` event. Its role `MyWriter` inherits from `Writer`, because the local choices in the latter have been eliminated in the former. Therefore component `MyProducer` inherits from `Producer`. 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 5 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 the added behavior is globally decided and won't interfere with roles that don't use it. Thus, component `Stack` inherits from `Buffer` (in fact, it implements the buffer, because we hadn't provided a specification for `Buffer` yet).

These relations of inheritance among components can be checked in LEDA, and they ensure that we could build a Producer/Consumer system using the components in Figure 2 and Figure 5. 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.*

6.3 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 any other one which inherits from it. So, we can build an instance of the Producer/Consumer architecture which uses the `Stack` in Figure 5 as buffer:

```
prodConsStack:ProdCons(b:Stack);
```

or we can specify a particular Producer-Consumer system using the components described in Figure 4:

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

```
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).^o<item>.
      Stack(i,o,iq,oq,next,empty,writer)
  + [empty=TRUE][writer=FALSE]^oq.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 5. Specification of component `Stack`

with no need of checking the compatibility of the roles of these particular components. A single proof of role inheritance allow the replacement of a component in any system belonging to a family of related (but not identical) systems without verification of the compatibility of the modified attachments. Liveness properties of the architecture are ensured by static checking of inheritance, even in a scenario of dynamic, i.e. at run-time, binding.

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 in architecture instantiation. This property addresses component and architecture reuse in the same way as data polymorphism does in the object-oriented paradigm.

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

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 formal specification of dynamic systems.

Using LEDA, the specification of roles as interfaces of components is enough in order to verify the composability of the system from the compatibility of each role-to-role attachment. Besides, the specification of the components of the system 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 pattern framework which may be instanced in order to obtain particular systems which maintain the architectural properties of the pattern.

Now, we are interested in verifying the practicability of our approach, using it in the specification of real-world systems. In particular, we are developing the specification in LEDA of an FTP server, from which a preliminary version can be found in [Canal 96].

8. References

[Allen 94] 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>

[Canal 96] C. Canal, E. Pimentel and J.M. Troya. "Software Architecture Specification with π-calculus". *Jornadas de Trabajo en Ingeniería del Software*, Sevilla, 1996. Available at: <http://www.fie.us.es/pub/lsi/jids>

[Canal 97] C. Canal, E. Pimentel and J.M. Troya. "A Formal Definition of Compatibility and Inheritance for Software Architectures". *Technical Report*. University of Málaga, 1997.

[Engberg 86] 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.

[Garlan 95] 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.

[Inverardi 95] 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.

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

[Milner 89] R. Milner. *Communication and Concurrency*, Prentice Hall, 1989.

[Milner 92] 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.

[Nierstrasz 95] O. Nierstrasz. "Requirements for a Composition Language". *Proc. of ECOOP'94*, LNCS 924, Springer Verlag, 1995, pp. 147—161.

[Pree 95] W. Pree. *Design Patterns for Object-Oriented Software Development*, Addison-Wesley/ACM Press, 1995.

[Pree 96] W. Pree. *Framework Patterns*, SIGS Publications, 1996.

[Sangiorgi 93] 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>

[Shaw 95] 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.

[Victor 94] B. Victor. "A Verification Tool for the Polyadic π-calculus". *Licentiate thesis*, Department of Computer Systems, Uppsala University, Sweden, May 1994.