Specification and Refinement of Dynamic Software Architectures

Calos Canal, Ernesto Pimentel[1], José M. Troya
Depto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain
E-mail: {canal, ernesto, troya}@lcc.uma.es

Key words: Software architecture, architecture description languages, π-calculus, compatibility, inheritance of behaviour, prototyping

Abstract: Several notations and languages for software architectural specification have been recently proposed. However, some important aspects of composition, extension, and reuse deserve further research. These problems are particularly relevant in the context of open systems, where system structure can evolve dynamically, either by incorporating new components, or by replacing existing components with compatible ones. Our approach tries to address some of these open problems by combining the use of formal methods, particularly process algebras, with concepts coming from the object-oriented domain. In this paper we present LEDA, an Architecture Description Language for the specification, validation, prototyping and construction of dynamic software systems. Systems specified in LEDA can be checked for compatibility, ensuring that the behaviour of their components conforms to each other and that the systems can be safely composed. A notion of polymorphism of behaviour is used to extend and refine components while maintaining their compatibility, allowing the parameterisation of architectures, and encouraging reuse of architectural designs.

1. INTRODUCTION

The term software architecture (SA) has been recently adopted referring to the discipline of Software Engineering that deals with the description,

[1] This work was funded in part by the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) under grant TIC98-0445-C03-03.

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: 10.1007/978-0-387-35563-4_35

verification, and reuse of the structure of software systems (Shaw and Garlan, 1996). At the level of abstraction of SA, software is represented as a collection of computational and data elements, or components, interconnected in a certain way, and it is at this level where the structural properties of software systems are naturally addressed. SA pays special attention to the interaction among components, instead of the internal computations of these components.

The significance of explicit architectural specifications is widely accepted. First, they raise the level of abstraction, facilitating the description and comprehension of complex systems. Second, they increase reuse of both architectures and components (Shaw and Garlan, 1995). However, effective reuse of a certain architecture often requires that some of its components can be removed, replaced, and reconfigured without perturbing other parts of the application (Nierstrasz and Meijler, 1995). These aspects are particularly relevant when dealing with open distributed systems, whose architecture evolves dynamically, and consistency has to be guaranteed for every substantial change produced on the system. In the context of SA, consistency must be analysed in terms of the compatibility between components, since system performance depends on the correct interaction among them.

Although object-orientation can be applied to all levels of software design, in SA 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 an architecture (Nierstrasz, 1995). However, most concepts coming from the object-oriented paradigm can be applied to SA. Particularly, in this work we address the application of inheritance, parameterisation, and polymorphism to the specification of software architectures.

A number of Architectural Description Languages (ADLs) have been already proposed. ADLs address the need for expressive notations in architectural design, trying to provide precise descriptions of the *glue* for combining components into larger systems. Despite the proposed notations are useful for the description of complex software systems, most of them are not formally based, which prevents the analysis and proof of the properties of the systems and architectures described (Abowd et al., 1993). In addition, several significant issues, such as specification of dynamic systems, architecture parameterisation and refinement, are not usually addressed. In Section 9 we compare our approach with other related works, particularly Wright and Darwin, while an exhaustive comparison on the characteristics of some outstanding ADLs can be found in (Medvidovic and Rosenblum, 1997).

In order to avoid some of these limitations, our interests focus on the application of formal methods to SA. Formal specifications have a precise meaning derived from the semantics of the notation used, and validation tools can be developed to prove properties of the systems specified. To this effect, process algebras are widely accepted for the specification of software systems, which can be checked for equivalence, deadlock freedom, and other interesting properties.

Dynamic architectures are those which describe how components are created, interconnected, and removed during system execution, and which allow run-time reconfiguration of their communication topology. Formal specification of such systems requires the use of an adequate formalism. In particular, we propose the use of the π–calculus (Milner et al., 1992), a simple but powerful process algebra which can express directly *mobility*, allowing the specification of dynamic systems in a very natural manner. However, the π–calculus is a low level notation, which makes difficult its direct application to the specification of large systems.

This was our original motivation for the development of LEDA, an ADL which embodies mechanisms of inheritance and dynamic reconfiguration. The language is structured in two levels: *components*, representing system parts or modules, and *roles*, which describe the observable behaviour of components. Roles are written in an extension of the π–calculus, thus allowing the specification of dynamic architectures. Each role describes the protocol that a component follows in its interaction with other components. In turn, components are described as composed of other components. The structure or architecture of a component is indicated by the relations among its subcomponents, which are expressed by a set of *attachments* or connections among the roles of these subcomponents.

LEDA differs from other ADLs in that it makes no distinction, at the language level, between components and connectors, i.e. connectors are specified as a special kind of components. This allows the language to be more simple and regular, and does not impose a particular compositional model in the description of software architectures.

Since the semantics of LEDA is written in terms of the π–calculus (Canal et al., 1998b), specifications can be both executed, allowing architecture prototyping, and analysed. In this sense, it is possible to determine whether a system is safely *composable*, i.e. whether its components present compatible behaviour and can be combined to form the system. This kind of analysis has been traditionally limited to interface conformance, but we are also interested in determining whether the *behaviour* of a component is compatible with that of its environment. On the other hand, component reuse would be encouraged if we could check whether a certain existing component can be used in a new system where a similar behaviour is required. Again, the intuitive notion of compatibility arises. We have formalised compatibility of behaviour in the context of π–calculus (Canal et al., 1998a), ensuring that compatible roles are able to interact successfully until they reach a well-defined final state. Architectures written in LEDA are tested for compatibility in each of the attachments among roles of their components. Compatibility does not require that the components involved have strictly complementary behaviour, since we usually want to connect components which match only partially.

Reuse of existing software components would be promoted if we had a way for adapting a component to an interface which is not compatible with its own interface. This is what LEDA adaptors are made for. Adaptors are small elements, similar to roles and also written in π–calculus, which are able to communicate successfully components whose behaviour is not compatible.

Our approach is completed with mechanisms of inheritance and parameterisation for roles and components which ensure that compatibility is preserved. A child component inherits its roles from its parents, while redefinition of behaviour is restricted by several conditions which ensure the maintenance of compatibility. Thus, we can replace safely a component in an architecture with any other component which inherits from the former. This gives place to a mechanism of architecture instantiation, by which a software architecture can be considered as a generic framework, which can be partially instanced and reused as many times as needed. Component frameworks derive from the idea of design patterns, and they represent the highest level of reusability in software development: not only source code of components, but also architectural design is reused in applications built on top of the framework (Pree, 1996). In this sense, LEDA specifications can be considered as generic architectural patterns or frameworks which can be extended and reused, adapting them to new requirements (Canal et al., 1997).

Although specification certainly plays an important role during system design and prototyping, the final goal of software design is to obtain real executable applications. LEDA specifications are also used for the creation, interconnection and deletion of components on an executable distributed platform. Combining the capabilities of prototyping and execution of LEDA, it is possible to simulate the execution of partially implemented systems. Hence, system development can be done gradually, providing a smooth transition from specification to implementation.

The structure of this paper is as follows. First, we describe briefly the π–calculus and the notation we use for specifying roles with it. Then, Sections 3 and 4 deal with the specification of components, roles and attachments in LEDA. Next, in Section 5 we discuss how our approach addresses architecture prototyping and validation, while Section 6 deals with component and role inheritance, and also addresses the topic of architecture refinement. Section 7 shows how non-compatible components can be interconnected using adaptors. All the notions introduced in these sections are illustrated by several examples. Finally, Section 8 discuss briefly how LEDA specifications can be used in order to derive executable applications from an architecture. We conclude comparing our approach with some related proposals.

2. THE π–CALCULUS

The π–calculus, developed by Milner as a successor of CCS, is specially suited for the description of dynamic systems, in which components are created and interconnected during system execution, because it permits direct expression of mobility. Mobility is achieved in π–calculus by the transmission of channel names as arguments or *objects* of messages. When a process receives a channel name, it can use this channel as a *subject* for future transmissions. This allows an easy and effective reconfiguration of the system. In fact, the calculus does not distinguish between channels and data, all of them are generically *names*. This homogeneous treatment of names is used to construct a very simple but powerful calculus. In contrast, π–calculus is a low level notation, and its use in industrial-size problems would be tedious and difficult.

LEDA embodies the π–calculus for specifying the roles which describe the behaviour of components. Roles are described in LEDA as processes, using a syntax which derives from the original notation of the π–calculus, adding some syntactic sugar to obtain more friendly specifications. Let *P,Q,...* range over processes, and *a,b,c,...* range over names. Sequences of names are abbreviated using tildes ($\tilde{a}$). Then, processes are built from names and processes as follows:

$$P ::= \mathbf{0} \mid \tau.P \mid x!(\tilde{o}).P \mid x?(\tilde{a}).P \mid (x)P \mid [x=z]P \mid P|Q \mid P+Q \mid A(\tilde{a})$$

Empty or inactive behaviour is represented by ***0***. Silent transitions, given by τ, model internal actions. Thus, a process $\tau.P$ will eventually evolve to *P* without interacting with its environment. An output-prefixed process $x!(\tilde{o}).P$ sends the names $\tilde{o}$ (objects) along name *x* (subject) and then continues like *P*. An input-prefixed process $x?(\tilde{a}).P$ waits for some names $\tilde{a}$ to be sent along *x* and then behaves like $P\{\tilde{o}/\tilde{a}\}$, where $\{\tilde{o}/\tilde{a}\}$ is the substitution of $\tilde{a}$ with $\tilde{o}$.

Restrictions are used to create private names. Thus, in *(x)P*, the name *x* is private to *P*. Private names can be exported to other processes simply by sending them as objects of output actions, as in *(z)x!(z)*. A match *[x=z]P* behaves like *P* if the names *x* and *z* are identical, and otherwise like ***0***.

The composition operator is defined in the expected way: $P \mid Q$ consists of P and Q acting in parallel. Summation is used for specifying alternatives: $P + Q$ may proceed to P or Q. The choice can be locally or globally taken. In a global choice, two processes agree synchronously in the commitment to complementary actions, as in

$$(\ldots + x!(\tilde{o}).P + \ldots) \mid (\ldots + x?(\tilde{a}).Q + \ldots) \rightarrow P \mid Q\{\tilde{o}/\tilde{a}\}$$

On the other hand, local choices are expressed combining the summation operator with silent actions. Hence, a process like $(\ldots + \tau.P + \tau.Q + \ldots)$ may proceed to P or to Q with independence of its context. We use local and global choices to state the responsibilities for action and reaction.

Finally, $A(\tilde{a})$ is an agent with names $\tilde{a}$. Each agent identifier A is defined by an unique equation: $A(\tilde{a}) = P$. The use of agents allows modular and recursive definition of processes.

Some examples of processes written in π–calculus can be found in the following sections, but for a detailed description of the calculus, including its transition system, we refer to (Milner et al., 1992).

3. COMPONENTS AND ROLES

LEDA is an ADL for the description and validation of structural and behavioural properties of software systems. The language is structured in two levels: *components* and *roles*. Components represent software pieces or modules, each one providing a certain functionality while roles describe the behaviour of components and are used for architecture validation, prototyping, and execution.

3.1 Components

LEDA distinguishes between component classes and instances, and provides mechanisms for the extension and parameterisation of components. The specification of a component class consists of three main sections: *(i)* **interface**, consisting of several role instances; *(ii)* structure or **composition**, consisting of several component instances; and *(iii)* **attachments**, which contains a list of connections which indicate how the component is built from its parts.

The interface of a component is described as a set of role instances, which specify the behaviour of the component from the point of view of each other component that interacts with it. Each role is a partial abstraction representing both the behaviour that the component offers to its environment, and the behaviour that it requires from those connected to it.

LEDA distinguishes between role classes and instances, and provides constructions for the extension and derivation of roles.

For instance, consider a file transmission between two components, named *Sender* and *Receiver* respectively. Component *Receiver* plays the role of *reader*, receiving the data which is sent by *Sender*, which acts as *writer* (*Figure 1*).

```
component Sender {              component Receiver {
   interface                       interface
      writer : Writer;                reader : Reader;
}                               }
```

Figure 1: Components Sender and Receiver

3.2 Specification of component's behaviour

Traditionally, interface description has been limited to the signature of the methods that a component imports and exports, or the messages that it can send or receive. However, our goal is to describe the observable behaviour of components, that is, how they react to external stimuli, and how input and output stimuli are related. This behaviour is described by the roles that form the interface of the component. Roles are specified as processes in the π–calculus.

Roles *Writer* and *Reader* in *Figure 2* specify the protocol of interaction between the components *Sender* and *Receiver*, i.e. they describe how these components behave in order to perform a successful data transmission. Data is transmitted matching two complementary actions *w!(data)* and *w?(data)*. As indicated by the use of local choices, the responsibility for action falls in the *Writer* part, which knows when the file has been completely transmitted, and sends an event *wq!()* (writer quits), while the *Reader* must be able to react to both *Writer* actions.

```
role Writer(w,wq) {                      role Reader(w,wq) {
 spec is                                  spec is
      τ.(data)w!(data).Writer(w,wq)            w?(data).Reader(w,wq)
    + τ.wq!().0;                             + wq?().0;
}                                        }
```

Figure 2: Roles Writer and Reader, from components Sender and Receiver

3.3 Composites

Components can be either simple or composite. A composite contains several subcomponents which are instances of other component classes. Any software system can be described as a composite. Thus, the syntax of LEDA does not distinguish between components and systems or architectures. As we have shown, simple components are described by the roles of their interfaces, but for composites, we must also describe their internal architecture. This architecture is the result of the interconnection or attachment of several subcomponents. The specification of composites in LEDA will be shown by means of a set of examples of increased complexity, describing a family of systems following a Client/Server architectural pattern.

Consider first a very simple Client/Server system in which the *Client* requests services from the *Server* (*Figure 3*). Both the *Client* and the *Server* are composites which contain an unbound array of service components. Role *request* describes the behaviour of the *Client*, while role *serve* describes that of the *Server*. When receiving a request, the *Server* creates a *service* component with the statement **new**. Then, the reference to the service is transmitted to the *Client* through the private link *reply*. Notice that the type of the component *service* is not indicated, but is declared of a generic type **any**, allowing future refinement of the Client/Server architecture, as will be shown in Section 6, for providing different kinds of services. The name *n* is used in the role *serve* for taking account of the number of requests received, which will be also used in a subsequent example.

```
component Client {
 interface
  request : Request(request) {
   spec is
    (reply)request!(reply).
       reply?(service).Request(request);
 }
 composition
  service[] : any;
}
```

```
component Server {
 interface
  serve : Serve(request) {
   names
    n : Integer := 0;
   spec is
    request?(reply).
     (new service)reply!(service).
      n++.Serve(request);
  }
 composition
  service[] : any;
}
```

Figure 3: Components Client and Server with their roles

4. ATTACHMENTS

The architecture of a composite is determined by the relations that its subcomponents maintain with each other. These relations are explicitly represented in LEDA by a set of attachments among the roles of these subcomponents. Attachments relate roles of several components, and they are specified in the composite which contains these components. Attachments are set when the corresponding components and role instances are created, possibly dynamically, and can be modified during system execution.

LEDA distinguishes among several kinds of attachments, which permit the specification of both static, reconfigurable, and dynamic software systems.

Static attachments are those which are never modified once they are set. For instance, recall the components *Client* and *Server* from *Figure 3*. We can specify our Client/Server architecture as a composite which contains both components and connects their roles using a static attachment (*Figure 4*). The symbol used for indicating the attachment is <>.

```
component ClientServer {
 interface none;
 composition
  client : Client;
  server : Server;
 attachments
  client.request(r) <> server.serve(r);
 }
```

Figure 4: A simple Client/Server system

On the other hand, *reconfigurable* attachments are used for architectures that present several configurations, i.e. those in which the interconnection patterns among components changes over time, and the roles connected depend on a certain condition. For instance, suppose that we have two *Server* components, and each request is assigned to one of them trying to balance their work load (*Figure 5*).

```
component ReconfigurableClientServer {
 interface none;
 composition
  client : Client;
  server[2] : Server;
 attachments
  client.request(r) <> if ( server[1].n <= server[2].n )
                        then server[1].serve(r)
                        else server[2].serve(r);
 }
```

Figure 5: A reconfigurable system, consisting of one Client and two Servers

Finally, *multiple* attachments describe communication patterns among arrays of components. Each pair of interconnected components may use private links in their communication, or these links may be shared by all the components involved. Thus, multiple attachments can be either shared or private. A shared attachment describes a 1:M communication channel, while private attachments establish multiple 1:1 communication channels.

For instance, consider a more realistic Client/Server system in which several *Clients* are connected to a pool of *Servers*. The composite *ServerPool* (*Figure 6*, left) contains an array of *Servers* whose roles are tied together using a multiple shared attachment (represented by the '*' in the left part of the attachment), and exported as a single role *serve* (role exportation is described below). Each request will be served by one of the *Servers* non-deterministically. On the other hand, the attachment between the *Clients* and the *ServerPool* is also multiple (*Figure 6*, right), and all clients share the link *r* through which they request services. Notice that mobility is used to establish private *reply* links for each request, though all the *Clients* are connected to the *ServerPool* using a single *request* link. Such an example can be hardly specified using formalisms like CSP (and consequently with CSP-based ADLs like Wright), which shows the richer expressiveness of the π–calculus when compared with other process algebras.

```
component ServerPool {                component MultipleClientServer {
 interface                                    interface none;
  serve : Pool;                        composition
 composition                            client[] : Client;
  server[] : Server;                    pool    : ServerPool;
 attachments                           attachments
  server[*].serve(r) >> serve(r);       client[*].request(r) <> pool.serve(r);
 }                                     }
```

Figure 6: A Client/Server system, with multiple clients and a pool of Servers

An additional form of attachment is that of role exportation. Usually, when dealing with a composite, some of the roles of its components are not used for the interconnection of these components, but to form the interface of the composite. Thus, we say that these roles are exported by the composite, which is indicated in LEDA using the operator >> instead of <>. We have already used this mechanism in *Figure 6*, left, where the roles of the *Servers* were exported to form the interface of the *ServerPool*.

5. ARCHITECTURE PROTOTYPING AND VALIDATION

The specifications written in LEDA can be used for prototyping. Attachments have a formal semantics (Canal et al., 1998b) which allows the derivation of π–calculus prototypes from architectural specifications. These prototypes can be executed using a π–calculus interpreter like the MWB (Victor, 1994). Thus, specifications can be tested at an early stage of the development process, checking their conformance with system requirements.

Apart from description and prototyping, LEDA specifications also serve for validation purposes. In particular, for determining whether a system is consistent, i.e. whether the behaviour of its components is compatible.

As we usually want to connect components that match only partially, the relations of bisimilarity customarily used in process algebras are not well suited for our purposes. Thus, we have defined a relation of role compatibility in the context of π–calculus. A formal definition of compatibility and its properties is out of the scope of this paper, but it can be found in (Canal et al., 1998a). A proof of compatibility for every system attachment using this relation ensures that the corresponding components will be able to interact safely until they reach a well-defined final state. Thus, if a software system is built according to the specifications of the architecture, no failure will arise from the interaction in any attachment between its components.

Obviously, local analysis of compatibility cannot ensure that the whole system is deadlock-free, since deadlock could arise from the global interaction of a set of components whose roles are compatible. However, compatibility serves for determining whether two components can be composed or plugged into each other, guaranteeing that the connector <> is safe. We consider that a system is consistent when each attachment in its architecture connects compatible roles, indicating behavioural conformance of the corresponding components. On the other hand, a failure detected when analysing an attachment stands for a mismatch in the behaviour of the corresponding components, usually leading to a system crash.

6. EXTENSION AND REFINEMENT

6.1 Extension of roles and components

In order to promote effective reuse of both components and architectures, a mechanism of redefinition and extension for roles and components is required. In the object-oriented paradigm, reuse is achieved by inheritance and polymorphism. Data polymorphism is defined as the capability of an identifier to point or refer to instances of different classes, while inheritance refers to a relation among object classes by which an heir class inherits the features (methods and attributes) of its parent classes. Heirs can extend their parents by adding new features, and they may also redefine some of the inherited features, usually under certain restrictions. Inheritance is a natural precondition for polymorphism, since it ensures that heirs will have at least the same features than their parents, and that they can replace them safely.

A relation of inheritance will 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 features (i. e. the signature of its roles), but it also includes the behavioural patterns described in the roles. Thus, role redefinition and extension must be restricted in order to preserve the behaviour specified in the parent role. We have defined a relation of inheritance among roles in the context of π–calculus. This relation defines the restrictions for polymorphism of behaviour, allowing the replacement of a role by a derived version, while preserving compatibility. Role extension in LEDA can be formally validated. Again, we refer to (Canal et al., 1998a) for a formal definition of role inheritance and its properties.

Role extension can be used to *(i)* redefine, partially or completely, the parent role, giving a new specification for some of its agents; and *(ii)* extend a role, providing it with additional functionality. In both cases we must check, using the relation of inheritance, that the extended role is effectively an heir of the parent role.

For instance, consider the role *Serve* of *Figure 3.* Its behaviour can be extended allowing clients to query the number of requests solved by the server, which can be used for statistics.

```
role StatServe(request,statistics) extends Serve {
  adding
    statistics!(n).StatServe(request,statistics);
  }
```

Figure 7: An extension of role Serve

The notion of extension can be also applied to components. Derived components inherit their parent's specification, including roles, subcomponents and attachments. An heir component extends its parent by adding new roles, components, or attachments, or by redefining some of its parent's. In case of redefinition of a role or component, the redefined instance must be an heir of the original instance.

Component extension can be implicitly achieved by *architecture instantiation*, which indicates the replacement of a component instance in a composite with another one whose class extends that of the former. Architecture instantiation can be used for incremental specification, description of families of software products sharing a common architecture, and also for dynamic replacement of a component in a software system. The syntax of instantiation is as follows:

derivedComponent : ComponentClass[subcomponent : DerivedSubcomponentClass];

which means that *derivedComponent* is an instance of *ComponentClass* in which we have replaced its *subcomponent* (which let's suppose was declared of a certain *SubcomponentClass*) by an instance of *DerivedSubcomponent-Class*, where *DerivedSubcomponentClass* must be an heir of *Subcomponent-Class*.

When instancing an architecture, some of its attachments are modified, since some of its former components are replaced by derived versions. However, compatibility rechecking of the instanced architecture is not required, since role inheritance ensures the preservation of compatibility.

6.2 Architecture Refinement

Architectural descriptions can be used with different levels of abstraction during the development process. This property is commonly referred to as refinement. For example, we can begin with a high level specification of a system in which we describe only its top-level components, their interface, and how they are attached to construct the system. Then, refinement is applied to obtain a more detailed specification, by describing the internal structure or the behaviour of previously defined components, obtaining more complex specifications which come gradually closer to implementation. As we have seen, component extension is a useful mechanism for refinement, but other forms of refinement can be applied using LEDA.

In the Client/Server system in *Figure 6*, services were defined as generic components of type **any**. Thus, we have described an abstract Client/Server architecture which follows a simple protocol of requests and replies. We can obtain more specific architectures by describing the details of the service, i.e. describing the behaviour that both components follow during the service.

```
component ReceiverClient extends Client {
 interface
  request : RequestSenders(request) extends Request {
   spec is
    (reply)request!(reply).
     ( new receiver)reply?(service).RequestSenders(request);
  }
 composition
  receiver[] : Receiver;
  service[]  : Sender;
 attachments
  receiver[].reader(w,wq) <> service[].writer(w,wq);
}
```

Figure 8: Specialisation of a Client/Server, using Senders and Receivers

The *ReceiverClient* in *Figure 8* is a specialisation of the *Client* in *Figure 3*. Its role *request* is refined indicating that a component *receiver* is created each time the client requests a service. The *service* itself is refined, too, indicating that its type is now *Sender* instead of **any**, and a new attachment is included, connecting the roles of the *receiver* and the *service*. Components *Receiver* and *Sender* were specified in *Figure 1*, while its roles were described in *Figure 2*.

Hence, we have refined our Client/Server architecture, obtaining the description of a system in which the service provided is a file transmission. We can use the mechanism of architecture instantiation for obtaining an instance of the refined architecture:

refinedCS : MultipleClientServer[client : ReceiverClient, pool.server[].service[] : Sender];

Since role *RequestSenders* extends *Request*, compatibility with server's role *Serve* is ensured. On the contrary, the compatibility of the new attachment between the roles *Reader* and *Writer*, which was not present in the original architecture, must be checked.

7. ADAPTORS

Sometimes the behaviour of two components is not compatible, but these components can be adapted so they can collaborate with each other. This will be done using an adaptor, which acts as a glue allowing the construction of composites from components which are not strictly compatible. Adaptors are also used to modify the interface that a certain component exports to its environment.

Adaptors are specified in π–calculus, using the same syntax as for roles. However, roles describe the interface of a component, and they are declared in the interface section, while adaptors are mainly used as a glue to tie the components of a composite, and they are declared in the composition section.

In the preceding examples, servers are always prepared to receive requests, which is not a realistic assumption. The specification of a non-reliable server *NRServer* is shown in *Figure 9*, left. Observe how local choices, indicated by the combination of the sum operator and τ-transitions, specify that the *NRServer* may crash unexpectedly.

Obviously, the behaviour of our *NRServer* is not compatible with that of *Clients*, which suppose that servers are always willing to attend their requests. However, using a simple adaptor *restart* we can build a fault-tolerant server pool (*FTServerPool*, *Figure 9*, right). Each time an *NRServer* crashes it is restarted by the adaptor (in fact, it creates a new *NRServer*).

Thus, the adaptor modifies the observable behaviour of the pool of *NRServers*, and the combination of roles *NRServe* and the adaptor *Restart* provides an interface which can be proved as a refinement of role *serve* in *Figure 6*. Thus, *FTServerPool* extends *ServerPool*, and its behaviour is also compatible with role *Request*.

```
component NRServer {
 interface
  serve : NRServe(request,crash) {
   spec is
    τ.request?(reply).
     ( new service)reply!(service).
       NRServe(request,crash)
    + τ.crash!().0;
   }
 composition
  service[] : any;
 }
```

```
component FTServerPool extends ServerPool {
 composition
  server[] : NRServer;
  restart  : Restart(crash) {
   spec is
    crash?()( new server)Restart(crash);
   }
 attachments
  restart(e),server[*].serve(r,e) >> serve(r);
 }
```

Figure 9: A fault-tolerant pool of servers, built from non-reliable servers

Hence, we can instance the Client/Server architecture of *Figure 6* replacing its component *ServerPool* by an instance of *FTServerPool*:

ftcs : MultipleClientServer[pool : FTServerPool];

Compatibility with client's role *request* is ensured by inheritance, and there is no need to recheck the attachment between the server pool and the clients. Thus, we obtain a specialised version of the Client/Server system in which we use non-reliable servers, but maintaining the properties of the original architecture.

8. SYSTEM CONSTRUCTION AND EXECUTION

We have already discussed how LEDA specifications can be used for system validation and prototyping, but we can go one step further, and use them also for obtaining an executable system.

Using LEDA we can validate that each attachment in an architecture connects compatible roles. Our goal is now to translate this compatibility to the implementation level. First, each role is automatically translated into a state machine which encapsulates the behaviour of the corresponding component. These implementations of roles control the interaction of the corresponding components with the rest of the system. Thus, they are similar to IDL specifications, but augmented with the protocol that describes the behaviour of the components.

In turn, composites are responsible for the creation of components and for interconnecting their roles, following the communication patterns described in their attachments. Communication between roles is done using a process communication mechanism, (e.g. sockets).

Finally, components must be implemented using a programming language. Typically each component specification will be implemented as a class or group of classes using an object-oriented language. Each component is connected to its roles, through which it communicates with the rest of the system. When a component requires to invoke a method of another one, it invokes the corresponding method in its own role, which will contact the role of the other component in order to invoke the method.

Consider again the Client/Server system specified in *Figure 3*. Components *Client* and *Server* are implemented as classes, while their roles are translated into *RoleRequest* and *RoleServe* respectively (*Figure 10*, top).

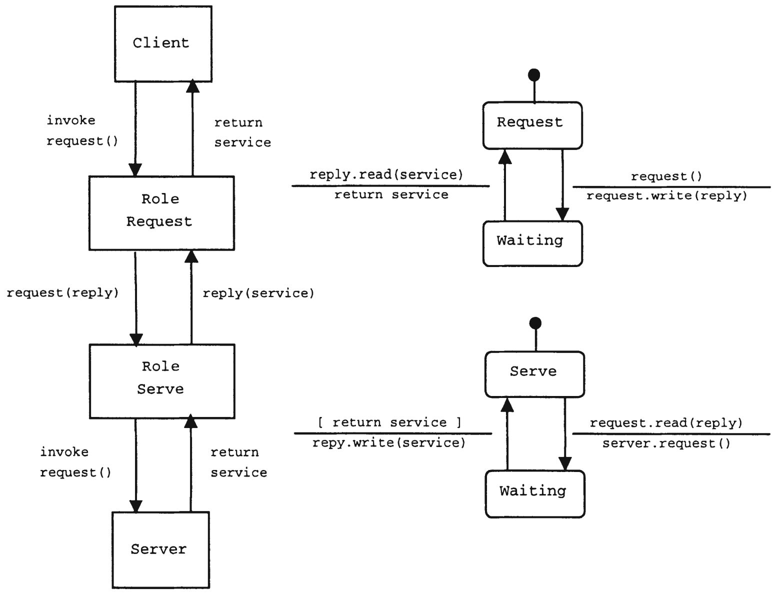

Figure 10: Implementation scheme of the Client/Server architecture

In order to obtain a service, the object *Client* invokes the method *request()* from its role *RoleRequest*. Then, *RoleRequest* sends the request to *RoleServe* through the appropriate channel. Next, *RoleServe* invokes the method *request()* from *Server*, and gets the service returned. The service is sent through a specific *reply* channel to *RoleRequest*, which in turns returns the service to *Client*. Thus, the implementations of *Client* and *Server* invoke or are invoked by their roles, but they don't know the location of the objects which finally receive the invocation, nor they are responsible for establishing or managing the communication channels indicated in the architecture.

This scheme for system implementation has several advantages. First, connections among components are encapsulated in the roles, which establish and modify them according to the interaction patterns specified in the architecture. Second, components are implemented as object classes that invoke or receive invocations of methods, but which are independent of the interaction mechanisms used in the architecture. Third, a component may have several implementations which can be interchanged without affecting the behaviour of the system.

9. DISCUSSION

In this paper we have presented LEDA, an ADL for the description of dynamic software architectures. In these systems, components interact following flexible patterns that can be modified during system execution. The basic unit in LEDA is that of components, which are represented by their interface, divided into a set of roles. These roles describe, using the π–calculus, the behaviour of the corresponding components. Software architectures are specified in LEDA as sets of components related by attachments between their roles. The semantics of components and attachments is given using the π–calculus, a well-known process algebra, which allows us to use this formalism for architecture prototyping and validation of properties like behavioural compatibility. LEDA roles and components can be extended, adapting them to new requirements, but maintaining the compatibility of the original roles. Analysis of compatibility and inheritance can be both automated, which leads to the development of tools for the analysis of the specifications. Formal validation of compatibility and inheritance encourage both software quality and reuse, determining whether some existing software components can be used to build a larger system.

In the last years several proposals related to the specification of software architectures have been presented. Although most of them are not formally founded, which limits their possibility of analysis, several works have already proposed the use of different formalisms for architecture specification.

A first formalisation of the notion of compatibility is described in (Allen and Garlan, 1997), where CSP is used for determining compatibility in the ADL Wright. However, formalisms like CSP or CCS do not seem appropriate for the description of evolving or dynamic structures. At most, CSP can be used in systems with a finite number of configurations, as it is shown in (Allen et al., 1998), but not in highly dynamic systems, where the π–calculus is best suited. Furthermore, Wright does not address aspects of component and role extension or refinement, nor of architecture simulation or execution.

Our approach differs from that of Allen and Garlan in other significant aspect: LEDA does not distinguish between components and connectors, nor between ports and roles. This distinction would complicate unnecessarily the language, specially the formalisation of compatibility and inheritance in π–calculus. Besides, we consider that the distinction between components and connectors does not scale properly, since composition would lead to mixed composites with free ports and roles which could not be considered either as components nor as connectors. For these reasons, connectors are described in LEDA as specific classes of components, their behaviour being described by roles.

The π–calculus has been used for describing the semantics of several computer languages. In fact, the operational semantics of the ADL Darwin (Magee and Kramer, 1996) is described using π–calculus, endowing this language with a mechanism of dynamic binding. However, type checking is restricted in Darwin to name matching, and the behaviour of components is not described, neither this language incorporates characteristics of extension or inheritance. On the contrary, our approach uses the π–calculus not only for semantics, but it integrates the calculus in the language. LEDA components and attachments are higher-level constructs that simplify the description of complex software systems, while LEDA roles take advantage of the expressiveness of the π–calculus for describing the behaviour of components. This allows us to state more precisely which are the relations between the components of a certain software architecture, and also to perform analysis of compatibility and inheritance.

The notions of component subtyping and inheritance are present in several other ADLs, and recent work of (Medvidovic et al., 1998), addresses description and verification of behavioural conformance using the Z notation. On the contrary, our approach describes component's behaviour using state machines, and addresses what they call *protocol conformance*.

We are currently working in the development of a Java run-time platform for LEDA, capable to use the information about component behaviour and architecture configuration present in the specifications to create, interconnect and remove the implementations of the components described using the language, thus obtaining executable applications.

Our future work will be the application of LEDA to the specification of different industrial software systems, in order to determine the need for new forms of interaction in the language. Another task will be the development of supporting tools, such as graphic editors or validation tools. All these tools should hide the difficulties inherent to the formal foundations of the language, making easier the specification of software systems in LEDA to those not acquainted with formal methods.

REFERENCES

Abowd, G., Allen, R., and Garlan, D. (1993). Using style to understand descriptions of software architecture. In *Proc. ACM FSE'93.*

Allen, R., Doucence, R., and Garlan, D. (1998). Specifying and analyzing dynamic software architectures. In *Proc. ETAPS'98*, Lisbon.

Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology.*

Canal, C., Pimentel, E., and Troya, J. (1997). On the composition and extension of software systems. In *Proc. of FSE'97 FoCBS Workshop*, pp. 50–59, Zurich.

Canal, C., Pimentel, E., and Troya, J. (1998a). Compatibility, inheritance and extension of π–calculus agents. Technical Report LCC-ITI-98-13, Computer Science Dept., Universidad de Málaga. http://www.lcc.uma.es/~canal/LCC-ITI-98-13.

Canal, C., Pimentel, E., and Troya, J. (1998b). π–calculus semantics of an architecture description language. Technical Report LCC-ITI-98-17, Computer Science Dept., Universidad de Málaga. http://www.lcc.uma.es/~canal/LCC-ITI-98-17.

Magee, J. and Kramer, J. (1996). Dynamic structure in software architectures. In *Proc. ACM FSE'96*, pp. 3–14, San Francisco.

Medvidovic, N. and Rosenblum, D. (1997). Domains of concern in software architectures and architecture description languages. In Proc. USENIX Conf. on Domain-Specific Languages, Santa Barbara (USA).

Medvidovic, N., Rosenblum, D. and Taylor, R. (1998). A Type Theory for Software Architectures. Technical Report UCI-ICS-98-14. Dept. Information and Computer Science, University of California, Irvine.

Milner, R., Parrow, J. and Walker, D. (1992). A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77.

Nierstrasz, O. (1995). Requirements for a composition language. In *Proc. of ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, no. 924 in LNCS, pp. 147–161. Springer Verlag.

Nierstrasz, O. and Meijler, T. (1995). Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264.

Pree, W. (1996). *Framework Patterns.* SIGS Publications.

Shaw, M. and Garlan, D. (1995). Formulations and formalisms in software architecture. In van Leeuwen, J., editor, *Computer Science Today*, no. 1000 in LNCS, pp. 307–323. Springer Verlag.

Shaw, M. and Garlan, D. (1996). Software Architecture. Perspectives of an Emerging Discipline. Prentice Hall.

Victor, B. (1994). A verification tool for the polyadic π–calculus. Master's thesis, Department of Computer Systems, Uppsala University (Sweden).