Specification of Interacting Software Components: a Case Study

Carlos Canal Ernesto Pimentel Jos M. Troya
Depto. de Lenguajes y Ciencias de la Computacin,
ETSI Informtica, Universidad de Mlaga
E-mail: {canal,ernesto,troya}@lcc.uma.es

Abstract

The emerging technologies challenge software companies to assimilate new development techniques addressing the specific characteristics of open and distributed applications. In this sense, two recent trends in Software Engineering are Component-Orientation and Software Architecture. Both consider software systems as collections of interacting components, and address aspects of composition and reuse. However, there is a lack of languages and tools for specifying the structure and interactions among the components of a system, or for determining whether these components have compatible behavior. Formal methods, in particular process algebras, can be a solution to some of these problems. In this paper we present the application of LEDA, a language based on the π-calculus, to the specification and validation of a distributed auction system, a particular problem where the interaction among system components is essential.

Keywords: component-based software development, software architecture, formal methods, process algebras, π-calculus.

1 Introduction

The emerging technologies using the Web as a medium for global communication and trade involve the development of open and distributed applications. The specific characteristics of these software systems and their supporting platform have an enormous impact in the way software is produced and marketed. These applications have a complex and usually dynamic structure, formed by the composition of a number of interacting components. Software companies must assimilate new development techniques capable to cope with these characteristics while improving the quality of their products. In this field there are currently two close-related trends: Component-Orientation and Software Architecture.

The goal of component-oriented software development is to create a collection of reusable software components. Application development then becomes the selection, adaptation and composition of components rather than implementing the application from scratch [3]. Software reuse reduces costs and time in the development process and increases software quality and reliability. However, effective reuse of a software component often requires that some of its parts can be removed, reconfigured or specialized, to accommodate it to new requirements [9].

On the other hand, the term *software architecture* has been recently adopted referring to the structure of a software system, i.e. the way in which its components are combined. It is at this level where the description and verification of structural properties is naturally addressed [11]. Software architectures can be considered as *component frameworks* which can be partially instanced and reused as many times as required. Component frameworks derive from the idea of design patterns and they represent the highest level of reusability: not only single components, but also architectural design is reused in applications built on top of the framework [10].

Component-Orientation and Software Architecture techniques pay special attention to the interactions among components instead of the internal computations of these components. Although both approaches are currently deserving great interest, there is a lack of languages and tools for the specification and analysis of the structure of software systems. Usually, this structure is described in an informal manner, leading to problems in the development process [6]. However, it would be interesting to have mechanisms for determining whether a system is safely *composable*, i.e. whether its components present compatible behavior and can be combined to form the system. On the other hand, software reuse would be encouraged if we should be able to check whether a certain existing component can be used in a new system where a similar behavior is required.

The application of formal methods has proved specially useful at the early stages of the development process, both for specification and testing. Formal languages have a precise semantics and support several forms of reasoning. In particular, we are studying the application of the π-calculus, a well-known process algebra, to the specification of software architectures and components.

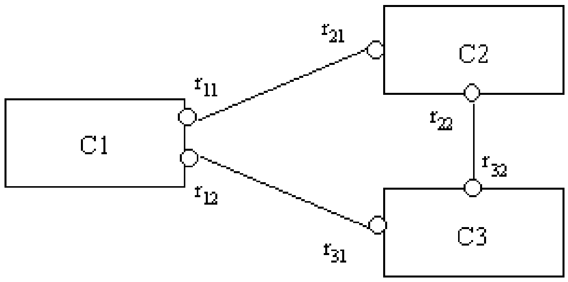
Although formalisms like π-calculus are hardly used in the development of industrial-size software systems, the development of higher-level languages and supporting tools will help to shorten the gap between formal foundations and their practical application. Examples can be found in emerging development tools integrating both formal and informal methods, like SDT [12], which combines SDL and the Unified Modeling Language (UML).

This is the approach we have taken in the development of LEDA [5], an architecture description language (ADL) for the specification of dynamic software systems. Components are first-class entities in LEDA, and are described from two points of view. Externally, components are described by their interface, divided into several units or *roles* which specify their observable behavior. On the other hand, components are internally described as composed of other components whose roles are conveniently attached to each other in order to obtain the required functionality. LEDA specifications can be considered as generic architectural patterns or frameworks which can be extended and reused, adapting them to new requirements. Since the language takes the π-calculus as its formal basis, executable prototypes can be derived from the specifications, which can also analyzed for compatibility, inheritance and other properties.

The structure of this paper is as follows. First, we describe the specification of software components in LEDA, introducing the notion of role as the elemental unit for describing the behavior of a component. Next, we show how roles are specified using the π-calculus, and how LEDA systems can be analyzed for compatibility. Then, Section 5 describes how components and roles written in LEDA can be extended or inherited. Section 6 shows the application of our approach to the specification of a distributed software system. We conclude discussing the most relevant characteristics of our work with respect to other proposals in its field, and presenting some future works.

2 Specification of software components

A software system is described in LEDA as a structure formed by several *components* (Figure 1, left) which interact following certain patterns of behavior. These components are system parts or modules, each one providing a certain functionality. Components are represented by their interface, which describe their observable behavior. Thus, internal computations and other implementation details, which are not relevant to the composition and interaction of components are omitted. LEDA




```
component X {
  composition
    c1,c2,c3;
  attachments
    c1.r11 <> c2.r21;
    c1.r12 <> c3.r31;
    c2.r22 <> c3.r32;
}
```

Figure 1: Several interconnected components

distinguishes between component classes and instances, and provides constructs for the derivation and instantiation of components.

Usually, a given component plays different *roles* in a system. Hence, we divide the interface of a component into a set of roles. Each role is a partial abstraction of the behavior of the component and represents both the functionality that the component offers to its environment and the functionality that it requires of those connected to it. A certain role can be found in many different software systems, laying the foundations for component reuse. LEDA distinguishes between role classes and instances and provides constructs for the extension and derivation of roles. Roles are written using an extension of the π-calculus described in Section 3.

Components can be either single or composite, containing several (single or composite) subcomponents. This allows modularity and hierarchical description. In fact, a whole software system can be considered as a composite component.

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 (Figure 1, right). We distinguish several kinds of attachments, which permit the specification of both static, reconfigurable, and dynamic software systems.

- **Static**. These are attachments which are set when the corresponding components are created and are never modified. For instance, consider a Client/Server system, in which client's role `request` is attached to server's role `provide` :

  ```
  client.request(service) <> server.provide(service);
  ```

- **Reconfigurable**. Reconfigurable attachments are dynamic. The roles connected depend on a certain condition. For instance, suppose now that a component Client is attached to a specific Server in a pool of two servers depending on their work load:

  ```
  client.request(service) <> if cond then server1.provide(service)
                                      else server2.provide(service);
  ```

- **Multiple**. Multiple attachments define communication patterns among arrays of components. They allow dynamic creation of components, which are conveniently attached when created. These attachments can be either shared or private to each pair of components. Shared attachments describe 1:M communication patterns, which implies broadcasted message emission, and their use will be shown in the Auction system in Section 6. On the other hand, private attachments establish multiple 1:1 communication patterns. For instance, suppose that after each Client's request, a Server creates a Daemon component for solving the request, allowing the Server to serve multiple Clients at the same time. Each Daemon is attached to the corresponding Client using a multiple attachment with private names:

```
client[].get(service) <> server.daemon[].serve(service);
```

Thus, the specification of a component in LEDA contains up to three sections: *(i)* `interface`, which describes the externally observable behavior of the component as a set of instances of role classes (defined inline or separately), *(ii)* `composition`, consisting of instances of component classes, and *(iii)* several `attachments` among the roles of these subcomponents.

```
component componentName {
  interface
    roleName :  roleClass; ...
  composition
    subcomponentName :  subcomponentClass; ...
  attachments
    subcomponentName.roleName <> subcomponentName.roleName; ...
  }
```

3 Specification of component's behavior

Process algebras are a good candidate for specifying the behavior of software systems. In particular, we have chosen the π-calculus because it permits direct expression of *mobility*. This makes this formalism specially suited for the description of dynamic systems, in which components and attachments are created and modified during system execution.

LEDA roles are written in an extension of π-calculus, adding some syntactic sugar to obtain more friendly specifications, but they can be almost directly translated into the original calculus.

Behavior is specified in LEDA by one or more *agents*, describing the behavior of the corresponding component. Roles interact with each other by means of their parameters which can be considered as bidirectional channels for the transmission of events and data.

```
role agentName(prmtrs) { ...
  spec is specificationOfBehavior;
  agent auxiliarAgentName(prmtrs) is specificationOfBehavior; ...
  }
```

Roles are written in LEDA using the following syntax:

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

Empty or inactive behavior is represented by $\mathbf{0}$. Silent transitions, given by τ, model internal actions. Thus, a role τ.P will eventually evolve to P without interacting with its environment. An output-prefixed role x!($\tilde{y}$).P sends the data $\tilde{y}$ (*objects*) along x (*subject*) and then continues like

P. An input-prefixed role $\mathtt{x?(\tilde{w}).P}$ waits for some data $\tilde{\mathtt{y}}$ to be sent along $\mathtt{x}$ and then behaves like $\mathtt{P}\{\tilde{\mathtt{y}}/\tilde{\mathtt{w}}\}$, where $\{\tilde{\mathtt{y}}/\tilde{\mathtt{w}}\}$ is the substitution of $\tilde{\mathtt{w}}$ with $\tilde{\mathtt{y}}$. Events are modeled by input and output actions without data transmission: `x!()`, `x?()`.

The composition operator is defined in the expected way: `P | Q` consists of `P` and `Q` acting in parallel. The sum operator 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 roles agree in the commitment to complementary transitions, matching synchronously complementary actions, as in

$$(\cdots + \mathtt{x!(z).P} + \cdots) \mid (\cdots + \mathtt{x?(y).Q} + \cdots) \xrightarrow{\tau} \mathtt{P} \mid \mathtt{Q}\{\mathtt{z/y}\}$$

On the other hand, local choices are expressed combining the summation operator with silent actions. Hence, a role like $(\cdots + \tau\mathtt{.P} + \tau\mathtt{.Q} + \cdots)$ may proceed to `P` or to `Q` with independence of its context. Local and global choices are used to state the responsibilities for action and reaction.

Restrictions are used to hide names. Thus, in `(^x)P`, `x` is private to `P`. Finally, $\mathrm{A}(\tilde{\mathrm{w}})$ is an agent with parameters $\tilde{\mathrm{w}}$.

Mobility is achieved in π-calculus by the transmission of channel names as arguments or objects of actions. When a process receives a channel name it can use this channel 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 called generically *names*. This homogeneous treatment of names is used to construct a very simple but powerful calculus. In contrast, π-calculus is a lower-level notation and its use in industrial-size problems would be tedious and difficult. This was our motivation for the development of a higher-order language.

Some examples of roles can be found in the following sections, but for a detailed description of π-calculus we refer to [8].

4 Role Compatibility

Apart from description, LEDA specifications serve also for validation. In particular, for determining whether the components of a system present compatible behavior. Furthermore, in order to enhance reusability, we must check if a certain existing component can be used in a new system where a similar behavior is required. Again, the intuitive notion of compatibility arises. Compatibility does not require that the components involved have strictly complementary behavior, since we usually want to connect components which match only partially, increasing the possibilities for reuse.

We have defined a relation of compatibility in the context of π-calculus. Although a formal definition is out of the scope of this paper, it can be found in [4], where it is also proved that compatibility ensures deadlock-freedom. Thus, compatibility serves for determining whether two components can be plugged into each other, and it lays the foundations for safe software composition.

LEDA specifications are tested checking their attachments for compatibility, which guarantees that the connector `<>` is safe. We consider that a system is *composable* when each attachment in its architecture connects compatible roles, which indicates full conformance in the behavior of the corresponding components. Compatible roles are able to interact without deadlock until they reach a well-defined final state. On the other hand, a deadlock detected when analyzing an attachment stands for a mismatch in the behavior of the corresponding components, usually leading to a failure or crash of the system.

Consider the following definition for roles `R11` and `R21` in Figure 1:

```
role R11(a,b)   { spec is τ.a?().R11(a,b) + τ.b?().R11(a,b);  }
role R21(a,b,c) { spec is a!().R21(a,b,c) + b!().R21(a,b,c) + c!().0;  }
```

Role `R11` indicates that the corresponding component `C1` may proceed to `a?().R11` or to `b?().R11`. Since `R21` presents a global choice with the complementary actions `a!()` and `b!()`, both roles are compatible. The presence of an unmatched action `c!()` in `R21` doesn't interfere with its compatibility with `R11`. This action is globally chosen in `R21`, and it is not considered in `R11`. Thus, it will never take place in the context of their composition. Action `c!()` may correspond with an operation offered by `C2` which is not used by `C1`. However, `R11` would not be compatible with a certain **role** `R21'(a,b,c)` { **spec is** `a!().R21(a,b,c) + b!().R21(a,b,c) +` τ`.c!().`**0**; }, since these roles may evolve to `a?().R11` or `b?().R11`, and `c!().`**0** respectively, which would deadlock.

The analysis of compatibility can be automated, which leads to the development of tools for testing the compatibility of software specifications written in LEDA.

LEDA specifications can be also used for prototyping. Since the semantics of the language is defined in terms of the π-calculus, prototypes are built by composing in parallel the roles which represent the components of a system, together with the attachments that relate these roles. Prototypes can be executed and tested using tools like MWB [13].

5 Extension of roles and components

Inheritance in LEDA permits that roles and components are extended and specialized. Inheritance encourages both reusability and incremental development, and it is a natural precondition for polymorphism, allowing the replacement of a component within a system by a specialized version which maintains some of the properties of the original one.

In our context, components are represented by the behavioral patterns described in their roles. In order to achieve behavioral polymorphism, an heir component must inherit the roles of its parents, possibly redefining some of them, though redefinition is restricted in order to ensure that compatibility is maintained. We have defined a relation of role inheritance in the context of π-calculus, and proved that it preserves compatibility [4]. The syntax for role extension in LEDA is as follows:

role *HeirRoleClass(names)* **extends** *ParentRoleClass* {
| **spec is** $NewSpec$;
| **redefining** $ParentAgent_1$ **as** $NewSpec_1$; ... $ParentAgent_n$ **as** $NewSpec_n$;
| **adding** $NewSpec_1$; ... $NewSpec_m$; }

Extension can be used to (i) redefine completely the parent role, giving a new specification; (ii) redefine some agents specified in the parent role; and (iii) extend a role, adding new behavior.

With respect to components, a component extends or inherits another one when all the roles of the latter are inherited by the former. If we replace a component of a system by one of its heirs, some of the attachments in which the parent component participated will be modified, but as inheritance preserves compatibility there is no need to recheck these attachments. A single proof of inheritance ensures compatibility in a family of software products.

In this sense, a system specified in LEDA may be considered as a generic architectural pattern or framework, whose subcomponents are formal parameters that may be instanced with components that inherit from them. Component instantiation is expressed in LEDA with the following syntax:

```
mysystem :  Architecture[ c1:C1, c2:C2];
```

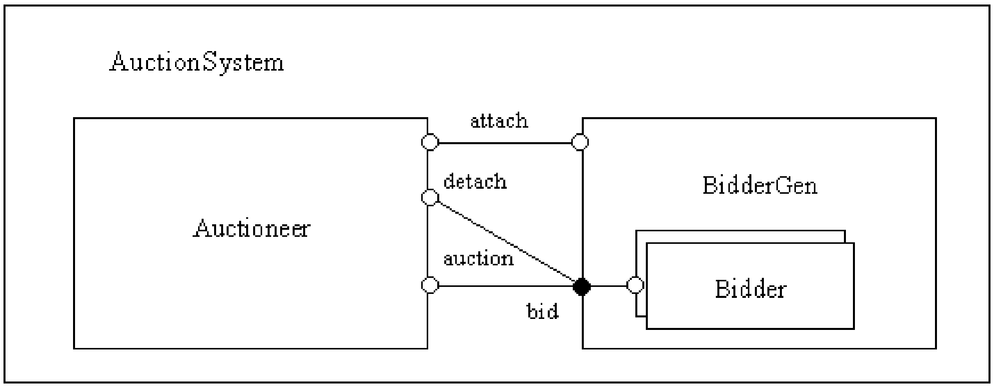


Figure 2: The Auction System

where `mysystem` is an instance of a certain `Architecture` in which subcomponents `c1` and `c2` have been replaced by instances of component classes `C1` and `C2`, respectively. `C1` and `C2` must be derived from the original classes defined for `c1` and `c2` in the `Architecture`, ensuring compatibility in the attachments of `mysystem`.

6 Case Study: a distributed auction system

We will show the applicability of our approach by means of an example of a distributed auction system. This could be a typical instance of system being developed for the Web.

6.1 Description

The system sells a set of items to the highest bid. The auctions are performed by an auctioneer and a number of bidders. At the beginning of each auction, the auctioneer informs the bidders about the characteristics of the item being auctioned and its initial price. While the auction is in progress, bidders can enter or leave the system, which is communicated to the auctioneer.

The auctioneer receives bids, which are accepted or rejected depending on the price offered. Both acceptance and rejection are notified to the corresponding bidder. When a bid is accepted, the rest of the bidders are informed of the updated price of the item.

When a bidder receives a price updating, it may react in three ways: *(i)* making a new bid with a higher price, *(ii)* sending an acknowledgement of the updated price to the auctioneer, which indicates that it refuses to bid at the moment, and *(iii)* leaving the application. Once a bidder has answered to a price updating, it is not expected to bid until it receives a new price.

The auction concludes when no more bids are offered. Then, the item is knocked down and sent to the corresponding bidder. The rest of the bidders are informed of the sale, and the auctioneer proceeds with the auction of another item.

6.2 System Specification

The system can be built up with a component `Auctioneer` connected to several components `Bidder` (Figure 2). In order to obtain a modular design, the interface of the auctioneer is divided into three roles: `attach` and `detach` are very general roles for the connection and disconnection of participants

```
component AuctionSystem {
  interface none;
  constants ACCEPT, REJECT;
  composition
    a : Auctioneer;
    b : BidderGen;
  attachments
    b.attach(attach) <> a.attach(attach);
    b.bid[](upd,bid,ack,sold,it,det) <> a.auction(upd,bid,ack,sold,it),
                                        a.detach(det);
  }

component Auctioneer {
  interface
    attach : Attach(att) { spec is att?().max++.Attach(att); }
    detach : Detach(det) { spec is det?().max--.Detach(det); }
    auction : Auction;
  var max : int;
  }

component BidderGen {
  interface
    attach : Attach(att) { spec is att!().Attach(att) | new bidder; }
    bid[] : Bid;
  composition
    bidder[] : Bidder;
  attachments
    bidder[].bid >> bid[];
  }

component Bidder {
  interface
    bid : Bid;
  }
```

Figure 3: LEDA specification of the Auction System

in any dynamic system, and they can be easily reused, while role `auction` refers to the specific behavior of the `Auctioneer`.

Dynamic connection of `Bidders` is simulated by `BidderGen`, which generates new `Bidder`s and attaches them to the `Auctioneer`. The attachment between the `Auctioneer` and `BidderGen` is static. The behavior of the `Bidders` is specified in their role `bid`. Thus, `BidderGen` exports an unbound array of roles `bid`. Since events referred to items' price and sale are broadcasted by the `Auctioneer` to the Bidders, all of them are attached to the same `auction` role, sharing channel names. Finally, role `bid` describes how `Bidders` detach the auction, so it is also connected to `Auctioneer`'s role `detach`. The specification of these components is shown in Figure 3.

6.3 Specification of behavior

The roles in Figure 4 specify the requirements of behavior that the components of the system must fulfill. Roles `attach` and `detach` in the `Auctioneer` indicate how this component is able to accept

```
role Auction(upd,bid,ack,sold,it)) {
  var val :  int;
  spec is
    val=INITIAL.upd[*]!(val).Auctioning(upd,bid,ack,sold,it) )

  agent Auctioning(upd,bid,ack,sold,it) is
     bid?(newval,back).
       ( τ.back!(ACCEPT).upd[*]!(newval).
           val=newval.Auctioning(upd,bid,ack,sold,it)
       + τ.back!(REJECT).Auctioning(upd,bid,ack,sold,it) )
   + ack?().Auctioning(upd,bid,ack,sold,it)
   + τ.sold[*]!().(^item)it!(item).Auction(upd,bid,ack,sold,it);
  }

role Bid(upd,bid,ack,sold,it,detach) {
  var id :  channel;
  spec is
     upd?(val).Deciding(upd,bid,ack,sold,it,det)
   + sold?().Bid(upd,bid,ack,sold,it,det);

  agent Deciding(upd,bid,ack,sold,it,det) is
     upd?(val).Deciding(upd,bid,ack,sold,it,det)
   + sold?().Bid(upd,bid,ack,sold,it,det)
   + τ.(^newval)bid!(newval,id).Waiting(upd,bid,ack,sold,it,det)
   + τ.ack!().Bid(upd,bid,ack,sold,it,det)
   + τ.det!().0;

  agent Waiting(upd,bid,ack,sold,it,det) is
    id?(result).( [result=REJECT].Bid(upd,bid,ack,sold,it,det)
                + [result=ACCEPT].Gaining(upd,bid,ack,sold,it,det) );

  agent Gaining(upd,bid,ack,sold,it,det) is
     upd?(val).Deciding(upd,bid,ack,sold,it,det)
   + sold?().it?(item).Bid(upd,bid,ack,sold,it,det);
  }
```

Figure 4: Roles of the Auction System

requests of connection and disconnection. The variable `max` indicates the number of participants currently connected and relates the three roles of the `Auctioneer`.

Role `auction` specifies how the `Auctioneer` manages the auction. First, the `Auctioneer` notifies the `Bidders` of the initial price of the item by broadcasting an event `upd[*]!(val)`. Then, it waits for acknowledgements (`ack?()`) or bids (`bid?(newval,back)`) informing the corresponding `Bidder` of acceptance or rejection. Notice how mobility is used here to gain access to a private reply channel (`back`). Accepted bids are notified broadcasting again an event `upd`. Finally, the `Auctioneer` knocks down the item after waiting for some time (here represented by a τ-transition) for a new bid. The sale is reported by broadcasting an event `sold()`, and the item is delivered with `it!(item)`.

On the other hand, role `Bid` has several states, each one represented by an agent. While in the initial state, the `Bidder` waits for a price updating. Then, it enters the `Deciding` state, where it can leave the auction (`det!()`), refuse to bid (`ack!()`) or make a new bid (`bid!(newval,id)`). When the latter occurs, the `Bidder` enters the `Waiting` state. If the bid is rejected (due to a higher

```
component FairAuctioneer extends Auctioneer {
  interface
    auction :  FairAuction;
  }

role FairAuction(upd,bid,ack,sold,it) extends Auction(upd,bid,ack,sold,it) {
  var n int;
  redefining Auctioning(upd,bid,ack,sold,it) as
     bid?(newval,back).
       ( τ.back!(ACCEPT).upd[*]!(newval).
           val=newval.n=0.Auctioning(upd,bid,ack,sold,it)
       + τ.back!(REJECT).n++.Auctioning(upd,bid,ack,sold,it) )
   + ack?().n++.Auctioning(upd,bid,ack,sold,it)
   + [n=max]sold[*]!().(^item)it!(item).Auction(upd,bid,ack,sold,it);
  }

component LtdBidder extends Bidder {
  interface
    bid :  LtdBid(upd,bid,ack,sold,it,det);
  }

role LtdBid(upd,bid,ack,sold,it,det) extends Bid(upd,bid,ack,sold,it,det) {
  redefining Deciding(upd,bid,ack,sold,it,det) as
     upd?(val).Deciding(upd,bid,ack,sold,it,det)
   + sold?().Bid(upd,bid,ack,sold,it,det)
   + τ.(^newval)bid!(newval,id).Waiting(upd,bid,ack,sold,it,det)
   + τ.ack!().Refusing(upd,bid,ack,sold,it,det)
   + τ.det!().0;

  agent Refusing(upd,bid,ack,sold,it,det) is
     upd?(val).ack!().Refusing(upd,bid,ack,sold,it,det)
   + sold?().LtdBid(upd,bid,ack,sold,it,det);
  }
```

Figure 5: Derived Auctioneer and Bidder components with their roles

bid from another `Bidder` since the last price updating), the `Bidder` returns to its initial state. However, if the bid is accepted, the `Bidder` enters state `Gaining`. Events `sold()` and `upd(val)` must be accepted in any state but `Waiting` (the `Auctioneer` is either obliged to accept or reject the bid), and they lead the `Bidder` to its initial state. When in `Gaining`, an event `sold()` means that the item has been knocked down to the `Bidder`, which gets it with `it?(item)`.

Using the relation of compatibility of [4], it can be proved that every attachment in the auction system connects compatible roles. This ensures that we can build the system, without fearing failures due to mismatch of behavior between its components.

6.4 System extension

Looking carefully to `Auctioneer`'s behavior we could see that it is not completely fair. It may decide to knock down the items at any time, without giving the participants chance to make a higher bid. A `FairAuctioner` can be developed storing the number of reactions to last price updating in a variable `n`, which is reset every time a new bid is accepted. Thus, only when `n=max` the item is knocked down. The `FairAuctioneer` has a more predictable behavior than the original `Auctioneer`, and it can be proved, using the relation of inheritance of behavior in [4], that extends it. Thus, we only have to declare the integer value `n`, and redefine agent `Auctioning` (Figure 5). Notice that the termination of the auctions is ensured, since `Bidders` must react to any price updating, either bidding, refusing to bid, or leaving the system.

We can also think of specific `Bidder`s which follow a particular bidding strategy. Provided their behavior conforms with that described in role `Bid`, they can participate in the auction. For instance, we can specify a `LtdBidder` (Figure 5) whose budget is limited to a certain amount of money. When the price of the item gets too high, it gives up bidding. Once again it can be proved that this component derives from the original `Bidder`. Hence, we can obtain a specialized version of our auction system just instancing it with the new components. Since they inherit the former, we can ensure that the system is compatible, with no need of recheck the attachments.

```
ltdFairAuction :  Auction[a:FairAuctioneer, b.bidder:LtdBidder];
```

7 Discussion

In this paper we have illustrated the application of LEDA to the specification and validation of dynamic software systems. In these systems components interact following flexible patterns which can be modified during system execution. LEDA provides constructs for specifying the architecture of software systems and describing the behavior of its components. The semantics of the language is expressed in terms of the π-calculus, which permits to obtain executable prototypes of the systems specified in LEDA. Besides, the relations of compatibility and inheritance defined in the context of π-calculus encourage both software quality and reuse, determining whether some existing software components can be used to build a larger system. These relations can be easily automated, which leads to the development of tools for the analysis of the specifications.

The main contributions of our work are: *(i)* LEDA takes advantage of the expressiveness of the π-calculus, particularly of mobility, applying it to the description of dynamic architectures, and *(ii)* as far as we are concerned, LEDA is the first ADL which addresses aspects of extension and inheritance of behavior among components. The verification of inheritance, allows safe replacement of components in a system.

In the last years several proposals related to the specification of software architectures and components have been presented [6, 7]. Although most of them are not formally founded, several papers have already proposed the use of different formalisms for architecture specification. In [7] the π-calculus is used for defining the semantics of Darwin, an ADL for the description of dynamic systems. However, type checking is reduced in Darwin to name equivalence, and aspects like compatibility and inheritance of behavior are not considered.

A first formalization of the notion of compatibility was proposed in [2], which uses CSP to determine the compatibility of asymmetric ports and roles. However, CSP does not seem appropriate for the description of evolving or dynamic structures. At most, it can be used in systems with a finite number of configurations, as it is shown in [1], but not in highly dynamic systems, where formalisms like π-calculus are best suited. Furthermore [2] does not address inheritance nor specialization issues. In [14], finite-state diagrams are used for the specification of what the authors call *protocols*, and relations of compatibility and protocol subtyping are also provided, though no higher-level language is proposed. While our approach arrives to similar results, the use of π-calculus presents several advantages. First, dynamic systems are quite naturally specified using this formalism. Second, LEDA roles and components are specified modularly, while in [14] messages are sent to a common *pool*, from which they could be retrieved by any component in the system; this being easily error-prone. Third, we use global and local choices to state the responsibilities for action and reaction, while they only take into account synchronous global behavior.

We have shown the applicability of our approach by means of a case study: a distributed auction in which bidders can enter or leave the system dynamically. Our future work will be the application of LEDA to the specification of different software systems, in order to determine the need for new forms of interaction or role attachment in the language. Another task will be the development of supporting tools, such as graphic editors, validation and prototyping tools, etc. 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

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

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, July 1997.

[3] J. Bosch. Adapting object-oriented components. In *Proc. ECOOP'97 Workshop on Component-Oriented Programming*, number 5 in General Publications. TUCS, 1997.

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

[5] C. Canal, E. Pimentel, and J. Troya. Specification and refinement of dynamic software architectures. In *Proc. 1st Working IFIP Conference on Software Architecture*, San Antonio (USA), February 1999. (accepted).

[6] D. Garlan and D. Perry. Special issue on software architecture. *IEEE Trans. on Software Engineering*, 21(4), April 1995.

[7] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proc. ACM FSE'96*, pages 3–14, San Francisco, October 1996.

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

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

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

[11] M. Shaw and D. Garlan. *Software Architecture. Perspectives of an Emerging Discipline*. Prentice Hall, 1996.

[12] Telelogic. Sdt reference manuals. http://www.telelogic.se.

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

[14] D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, March 1997.