Conformance and Refinement of Behavior in π-calculus

Carlos Canal   Ernesto Pimentel   José M. Troya

Depto. de Lenguajes y Ciencias de la Computación,
Universidad de Málaga, Spain
E-mail: {canal,ernesto,troya}@lcc.uma.es

Abstract

Process algebras are widely accepted for the specification of software systems. In particular, π-calculus addresses the description of dynamic systems, and permits their analysis for bisimilarity and other interesting properties. Though bisimilarity determines the equivalence of behavior, more flexible relations are required in the context of Software Engineering. In this paper we present a relation of compatibility in the context of π-calculus which formalizes the notion of conformance of behavior between software components. Our approach is enhanced with the definition of a relation of inheritance among processes. This relation preserves compatibility and indicates whether a process is a specialization or refinement of another one. The suitability of our approach is shown by its application to the field of Software Architecture.
Keywords: theory of concurrency, process calculi, software architecture

1 Introduction

Process algebras are widely accepted for the specification of software systems, in particular for communication protocols and distributed systems. One of the most popular and expressive formalisms in this family is the π-calculus [MPW92], which have been successfully applied in a number of different contexts. The π-calculus can express *mobility* in a direct manner. This makes it more adequate than other process calculi, like CCS or CSP, to specify dynamic systems, i.e. those which present an evolving communication topology.

The advantage of using an algebraic calculus is the capability to analyze the expected behavior of the system. Moreover, this analysis can be automated. Thus, it is usual to check processes for bisimilarity, in order to substitute parts in a system, or to justify a certain strategy for program transformation. However, some interesting properties of the components of a system are not related with bisimilarity. That is the situation in several fields of Software Engineering currently deserving active research, such as Software Architecture (SA) [SG96], where one of the properties that we may analyze is whether system components conform to each other. This has been traditionally limited to type checking of component interfaces, but we are also interested in checking whether the behavior of a component is *compatible* with that of its environment.

However, no specific relation among processes, able to capture the notion of compatibility, has been defined in π-calculus. Thus, we propose a relation of compatibility in this context. This relation ensures that two processes will be able to interact successfully until they reach a well-defined final state. It must be noticed that our relation is not an equivalence, nor a way of defining strictly complementary protocols of behavior. Our goal is to have a way of determining if two components are able to interact successfully. Thus, compatibility looks for the "safe" and flexible composition of software components.

On the other hand, bisimilarity supports the replacement of processes without affecting the behavior of the system. However, effective reuse of a software component often requires that some of its parts can be specialized to accommodate them to new requirements [NM95]. Again, current works on process algebras do not deal with this problem. In this way, our approach is completed with the definition of a relation of inheritance for the π-calculus. This relation preserves compatibility, promoting both incremental specification and reusability.

The aim of this paper is to describe the theoretical aspects of our work. Though it contains some examples, there is no much emphasis put on its applicability. However, the relations of compatibility and inheritance presented in this paper have been applied in the development of LEDA [CPT99a], an Architecture Description Language (ADL) based on the π-calculus, and a case study, a distributed auction system specified using this language, is contained in [CPT99b]. These papers show how our work address some open problems in the context of SA, but it should be noticed that the relations of compatibility and inheritance presented here are also applicable to the analysis of processes in general.

The rest of this work is structured as follows. In Section 2 we formalize the notions of role, attachment and architecture. Section 3 contains the definition of compatibility, and some results on how it ensures successful composition of the corresponding components. Proofs are only outlined, but they can be found in [CPT98]. Section 4 defines a compatibility-preserving relation of inheritance. We conclude discussing the relevance of our approach, comparing it with some related works. Finally, the appendix contains a short introduction to the π-calculus. We recommend to those not acquainted with this process algebra to read the appendix before the rest of this work.

2 Specification of Software Architectures

Software Architecture refers to the level of software design in which the system is structured as a collection of interacting components [GP95]. SA focuses on those aspects of design which cannot be suitably treated inside the modules that compose the system [SG96].

We propose the π-calculus for the specification of software architectures. Software systems can be described in π-calculus by composing the specifications of their components. However, this approach has two main drawbacks. First, the architecture of the system, which derives from the relations that each component maintains with the rest of the system, won't be explicitly represented. Second, state explosion would prevent the analysis of complex systems. Instead of that, we use partial interface specifications or *roles*, written in π-calculus, for describing the behavior of each component. Hence, system architecture is explicitly represented by describing attachments between roles, representing the interconnection of the corresponding components [CPT97]. Then, local analysis of each attachment is used to infer properties of the architecture. This reduces considerably the complexity, and justifies the use of the compatibility relation as an analysis tool.

Compatible roles will be able to interact successfully until they reach a well defined final state, indicating full conformance of the corresponding components. On the other hand, a deadlock detected when analyzing the compatibility of two roles stands for a mismatch in the behavior of the components, usually leading to a system crash.

In π-calculus practice, all terminating agents (i.e. those which cannot proceed performing one more transition) are referred to as *deadlock*, and they are equivalent to inaction. However, we must differentiate between agents which *are* the inaction, like $\mathbf{0}|\mathbf{0}$, and those which *behave* like the inaction, like $(a,b)(a(w).\mathbf{0}|\bar{b}y.\mathbf{0})$. Hence, structural congruence plays a significant role in distinguishing between successful and unsuccessful termination in the following definition.

Definition 2.1 (Success and Failure) *An agent A is a* failure *if there exists A' such that $A \Longrightarrow A', A' \stackrel{\tau}{\not\longrightarrow}$, and $A' \not\equiv \mathbf{0}$. On the contrary, A is* successful *if it is not a failure.*

Thus, we consider successful those agents which always proceed to the inaction without interacting with their environment, and failures those which would deadlock in the same conditions.

Now, we define roles as partial abstractions of the observable behavior of components, following the notion of abstraction in process algebras, as first established in [Bou87]. Definition 2.3 below is based on a *Hiding* operator ($/_{\mathcal{N}}$) similar to that described in [Mil89] for CCS, but adapted to the π-calculus:

Definition 2.2 (Hiding) *Let A be an agent and $\tilde{x} \subseteq fn(A)$, then*

$$A/_{\tilde{x}} = (\tilde{x})(A \mid \prod_{x \in \tilde{x}} Ever(x))$$

where $Ever(x) = x(y).(Ever(x)|Ever(y)) + (y)(\bar{x}y.(Ever(x)|Ever(y)))$

Definition 2.3 (Role) *An agent P is a* role *of a component Comp iff*

$$fn(P) \subseteq fn(Comp) \;\; \textit{and} \;\; P \approx Comp/_{(fn(Comp)-fn(P))}$$

where fn refers to the *free names* of a process (see the Appendix), i.e. the link names it uses for communicating with the rest of the system.

Thus, the specification of roles can be derived from that of the corresponding components by *hiding* some of their names. Any component action using hidden names will appear in the role as a silent action. Hence, when these actions are combined in the component with the sum operator, they will appear in the role as local choices, indicating that the component will take a decision on which of these actions occurs (local and global choice in π-calculus is discussed in the Appendix). For instance, consider the component $Comp(a, b, c, d) = a(x).\bar{b}x.\mathbf{0} + c(y).\bar{d}y.\mathbf{0}$. Following Definition 2.3, we can derive its role $P(b, d) = \tau.\bar{b}(x).\mathbf{0} + \tau.\bar{d}(y).\mathbf{0}$, where input actions from links a and c are abstracted by τ-transitions, while output actions on b and d are specified as local choices. The reason is that, from the point of view of another component connected only to links b and d, we cannot predict which of both output actions will take place.

Roles are defined modulo weak bisimulation. Given a component and some link names, several roles can be derived, but not all of them are correct. For instance, $P(a) = a(x).(c)c(y).\mathbf{0}$ is not a correct role of $Comp(a) = a(x).\mathbf{0}$, since action $a(z)$ leads $Comp$ to success but P to failure. Correctness is defined as follows.

Definition 2.4 (Correctness) *Let Comp be a component and $\mathcal{P} = \{P_1, \ldots, P_n\}$ a set of its roles. Comp is correctly specified by $\mathcal{P}$ iff*

1. $fn(P_i) \cap fn(P_j) = \emptyset \;\; \forall i, j = 1 \ldots n$ *s.t.* $i \neq j$
2. $Comp \equiv \mathbf{0} \iff \forall i \; P_i \equiv \mathbf{0}$
3. *If* $\exists \alpha \;.\; Comp \stackrel{\alpha}{\Longrightarrow} Comp'$, *then* $\exists i, P_i' \;.\; P_i \stackrel{\alpha}{\Longrightarrow} P_i'$, *and Comp' is correctly specified by* $\mathcal{P}' = \{P_1, \ldots, P_i', \ldots, P_n\}$

Hence, the roles in $\mathcal{P}$ must be disjoint, ensuring that they cannot synchronize, successful termination in *Comp* implies the same in the roles, and the interface of *Comp* must be completely specified by the roles in $\mathcal{P}$. Each role reflects a particular view of *Comp*, describing its behavior as it is seen from another component in the system connected to it.

a) `Buffer(put,get) = (null)( BufferSpec(put,get,null) | Null(null) )`

`BufferSpec(put,get,node) =`

`put(item).(new)( Node(new,item,node) | BufferSpec(put,get,new) )`

`+ get(return).node(item,next).`$\overline{\texttt{return}}$` item.BufferSpec(put,get,next)`

`Null(node) = `$\overline{\texttt{node}}$` `$\textsc{Null}$` node.Null(node)`

`Node(node,item,next) = `$\overline{\texttt{node}}$` item next.0`

b) `Storage(put) = put(item).Storage(put)`

c) `Retrieval(get) = get(return).(item)`$\overline{\texttt{return}}$` item.Retrieval(get)`

Figure 1: Component `Buffer` with its roles

Example 1. Figure 1 shows the specification of a `Buffer` (a), providing common `put` and `get` operations. The interface of the `Buffer` is divided into two roles: `Storage` and `Retrieval`, which are obtained by hiding respectively names `get` and `put`. Actions referring to hidden names would be represented by τ-actions in those roles, but these internal actions do not stand for local choices, so they can be omitted, obtaining by weak equivalence the roles (b) and (c). Notice that these roles don't indicate what is done with the items received; they only describe the behavior of our `Buffer`, as seen from outside, in a Producer/Consumer system. They represent the `Buffer` in its attachment to the roles representing the `Producer` and the `Consumer`, respectively. These attachments among roles are enough to describe and analyze the architecture of the system.

Thus, connections among system components are represented by *attachments* between their roles. In order to avoid synchronization between different attachments, free names must be conveniently restricted, as indicated in the definition below.

Definition 2.5 (Attachment) *The* attachment *of a set of roles* $\{P_i\}_i$ *is defined as*

$$(\cup_i fn(P_i))(\prod_i P_i)$$

The aim of the relation of compatibility described in Section 3 will be to establish the conditions that a set of attached roles must fulfill in order that their composition is successful. Finally, we can define an architecture, formed by the composition of several components, as a set of attachments between roles of these components.

Definition 2.6 (Architecture) *Consider a software system formed by several components* $\{Comp_j\}_{j=1}^{n}$. *Let* $\mathcal{R}_j = \{R_{j_i}\}_{i=1}^{n_j}$ *be the roles that specify correctly* $Comp_j$ $(j = 1 \ldots n)$. *Then, an* architecture *of the system is defined as a disjoint partition* Ψ *of* $\mathcal{R}oles = \cup_{j=1}^{n} \mathcal{R}_j$, *representing the attachments among roles that build the system from its components* $\{Comp_j\}_j$. *That is,*

$$\Psi = \{\mathcal{R}oles_1, \ldots, \mathcal{R}oles_m\}, \quad s.t. \;\; \mathcal{R}oles = \dot{\bigcup}_{k=1}^{m} \mathcal{R}oles_k$$

In order to simplify the results of the following sections, we will only consider *binary* architectures, where each $\mathcal{R}oles_k$ attaches only a pair of roles, but these results can be proved extensible to those architectures in Definition 2.6.

3 Role Compatibility in the π-calculus

The notion of compatibility introduced in this section determines when two agents or roles conform each other. A formal characterization is given in Definition 3.3, whose conditions ensure that no mismatch will occur when the roles are attached, representing the connection of the corresponding components.

First of all, compatible agents must be able to synchronize in, at least, one transition. This is a necessary condition for compatibility, preventing to consider compatible those agents like $a(x).\mathbf{0}$ and $\bar{b}y.\mathbf{0}$ which would fail when attached to each other.

Definition 3.1 *P* provides an input *for Q if* $\exists P', Q'$ *such that* $Q \stackrel{x(z)}{\Longrightarrow} Q'$ *and either* $P \stackrel{\bar{x}(z)}{\Longrightarrow} P'$ *or* $P \stackrel{\bar{x}y}{\Longrightarrow} P'$.

Definition 3.2 (Synchronizable roles) *P and Q are* synchronizable *if P provides an input for Q or Q provides an input for P.*

Definition 3.3 (Relation of (ground) compatibility) *A binary relation $\mathcal{C}$ on agents is a* semi-compatibility *if $P \; \mathcal{C} \; Q$ implies*

1. *if P is not successful then P and Q are synchronizable,*
2. *if* $P \xrightarrow{\tau} P'$ *then* $P' \; \mathcal{C} \; Q$,
3. *if* $P \xrightarrow{x(w)} P'$ *and* $Q \xrightarrow{\bar{x}y} Q'$ *then* $P'\{y/w\} \; \mathcal{C} \; Q'$
4. *if* $P \xrightarrow{x(w)} P'$ *and* $Q \xrightarrow{\bar{x}(w)} Q'$ *then* $P' \; \mathcal{C} \; Q'$

A relation $\mathcal{C}$ is a compatibility *if both $\mathcal{C}$ and $\mathcal{C}^{-1}$ are semi-compatibilities. The (ground) compatibility on agents,* $\dot{\diamond}$ *, is defined as the largest compatibility.*

The different treatment of silent and non-silent transitions in Definition 3.3 deals with global and local choices. Consider the roles $R_1 = a(x).\mathbf{0} + b(y).\mathbf{0}$ and $R_2 = \tau.a(x).\mathbf{0} + \tau.b(y).\mathbf{0}$, where input actions appear as global choices in R_1, while in R_2 the commitment to a particular action is locally decided. Thus, some roles which are compatible with R_1 are not compatible with R_2. Consider, for instance, $\overline{R_2} = \tau.\bar{a}u.\mathbf{0} + \tau.\bar{b}w.\mathbf{0}$. We have that $R_1 \; \dot{\diamond} \; \overline{R_2}$, since R_1 is able to accept any local choice in $\overline{R_2}$, but $R_2 \; \not\dot{\diamond} \; \overline{R_2}$, since they may proceed respectively to $a(x).\mathbf{0}$ and $\bar{b}w.\mathbf{0}$, which are not synchronizable.

Global choices which are not complementary in both roles are ignored in the definition of compatibility. Consider a role that presents an undesirable behavior when engaged in a certain action c, as $\overline{R_3} = \bar{a}u.\mathbf{0} + \bar{b}w.\mathbf{0} + \bar{c}\textsc{Error}$. Then we have both $R_1 \; \dot{\diamond} \; \overline{R_3}$ and $R_2 \; \dot{\diamond} \; \overline{R_3}$, as action c is not considered in those roles and the transition will never take place when composing $\overline{R_3}$ with R_1 or R_2. However, a role which could choose locally to engage in this action, like $\overline{R_4} = \bar{a}x.\mathbf{0} + \bar{b}y.\mathbf{0} + \tau.\bar{c}\textsc{Error}$ is not compatible with any of the roles above.

Definition 3.4 (Compatible roles) *P and Q are* compatible, *written $P \diamond Q$ if, for all substitutions σ that respect the distinction $fn(P) \cup fn(Q)$, we have that* $P\sigma \; \dot{\diamond} \; Q\sigma$.

In order to understand why we avoid substitutions equating free names, consider the roles $P = \bar{a}(x).\bar{x}.\mathbf{0}$ and $Q = a(y).y.\mathbf{0} + b(z).\mathbf{0}$. We have that $P \; \dot{\diamond} \; Q$ under most substitutions, but for $\{a/b\}$ we have that $P\{a/b\} \xrightarrow{\bar{a}(x)} \bar{x}.\mathbf{0}$, $Q\{a/b\} \xrightarrow{a(x)} \mathbf{0}$ and $\bar{x}.\mathbf{0} \; \not\dot{\diamond} \; \mathbf{0}$. This restriction set upon substitutions reflects the situation with real software components. If two components follow a certain protocol which rules their interaction, we must distinguish between the different channels or messages used in this protocol, if not communication would be impossible. Notice that, when roles are attached, their free names are restricted (see Definition 2.5), ensuring their distinction.

Our relation of compatibility is symmetric, and it presents some other desirable properties, related to compositionality, as shown in [CPT98]. However, it refers to complementary actions, so it does not satisfy other common properties like reflexivity or transitivity. In fact, compatibility is defined with the idea of having a safe and flexible way to connect processes, whereas relations like bisimilarity determine when processes are equivalent or simulate each other. For a detailed comparison between compatibility and bisimilarity we refer again to [CPT98].

Compatibility must ensure that the attachment of compatible roles is successful, indicating that no mismatch will occur from the interaction of the components involved. This is the aim of Proposition 3.5 below.

Proposition 3.5 *Let P and Q be compatible roles. Then we have that their attachment is successful.*

Proof. It can be directly derived from Definitions 2.5 and 3.3. □

Now, we can go onestep beyond, showing the effect of combining a component with roles compatible with its own roles.

Theorem 3.6 *Let Comp be a component with correct roles $\{P_i\}_i$. Let these roles represent Comp in its attachment to several other components $\{Comp_i\}_i$ in a certain system. Let Q_i be the role that represents respectively each $Comp_i$ in its attachment to Comp. Assume that $P_i \diamond Q_i$ for all i. Then we have that*

$$Comp \mid \prod_i (fn(Q_i) - fn(P_i)) Q_i$$

is successful.

Proof. Definition 2.4 ensures that the roles $\{P_i\}_i$ are disjoint, but some Q_i may have additional free names. Thus, we have to restrict them, ensuring the independence of the attachments of *Comp*. Then, from $P_i \diamond Q_i$ for all i and Definitions 2.3 and 2.4, we can derive the success of the composition of *Comp* with $\{Q_i\}_i$. □

This result refers to the composition of a component with compatible roles, but not with the corresponding components. In fact, if two components are attached we cannot derive that their composition is successful, since these components may be in turn connected to other components in the system. Furthermore, we cannot ensure either that the whole system is successful, since deadlock could arise from the global interaction of components whose roles are compatible. However, the compatibility of system attachments ensures that no system crash will arise from a mismatch in the behavior specified by the interfaces of the components. This is enough to prove the *composability* of a system. Now, we can define a *composable architecture* as a set of attachments between compatible roles.

Definition 3.7 (Composable Architecture) *Consider a binary architecture Ψ under the conditions of Definition 2.6. We say that Ψ is a* composable architecture *if, for every pair or roles $P, Q \in \mathcal{R}oles$, such that $\{P, Q\} \in \Psi$ we have that $P \diamond Q$.*

Hence, the composability of an architecture is determined by testing the compatibility of its attachments. For simplicity, we have restricted architectures to attachments between pairs of roles. However, a more general definition can be considered by extending Definition 3.3 to groups of agents.

4 Inheritance and Extension of Behavior

In this section we present a relation among roles for checking whether a certain existing component can be used in an architecture where another one appears.

This notion is related to the concept of inheritance in the object-oriented paradigm. Inheritance is a natural precondition for polymorphism, allowing the replacement of a component by a derived version, and it promotes both reusability and incremental development. Hence, our relation must preserve compatibility, and derived roles must inherit their parents' behavior, while redefinition and addition of behavior must be restricted in order to maintain compatibility. We have called this relation *role inheritance*. However, derived roles may also offer new globally chosen behavior, which we call in turn *role extension*. Since a compatibility-preserving relation of inheritance requires the fulfillment of several conditions, the relation will be introduced in several steps.

Definition 4.1 (Semantics-preserving inheritance) *A binary relation $\mathcal{H}$ on agents is a* semantics-preserving *inheritance if $R\mathcal{H}P$ implies*

1. *if $P \xrightarrow{\bar{x}y} P'$ then $\exists R'.\ R \xrightarrow{\bar{x}y} R'$ and $R'\mathcal{H}P'$*
2. *if $P \xrightarrow{x(y)} P'$, $y \notin n(P,R)$ then $\exists R'.\ R \xrightarrow{x(y)} R'$ and $\forall w, R'\{w/y\}\mathcal{H}P'\{w/y\}$*
3. *if $P \xrightarrow{\bar{x}(y)} P'$, $y \notin n(P,R)$ then $\exists R'.\ R \xrightarrow{\bar{x}(y)} R'$ and $R'\mathcal{H}P'$*

Assume that $P\diamond = \{Q : P \diamond Q\}$ is the set of processes which are compatible with P, and assume that $R\mathcal{H}P$. Then, the conditions above imply that any globally chosen transition in P must be preserved by R, maintaining R compatible with any agent $Q \in P\diamond$ which makes a local choice over one of these transitions. Hence, for roles $P = a(x).\mathbf{0} + \tau.b(y).\mathbf{0}$, $R = a(x).\mathbf{0}$, and $Q(a,b) = \tau.\bar{a}(u).\mathbf{0} + \bar{b}(v).\mathbf{0}$, we have that $P \dot{\diamond} Q$, $R\mathcal{H}P$, and $R \dot{\diamond} Q$. However, no condition is imposed over τ-transitions, thus local choices may be suppressed by the child. It will be shown that Definition 4.1 describes necessary but not sufficient conditions for inheritance, which can be extended as follows.

Definition 4.2 (Non-extensible inheritance) *A binary relation $\mathcal{H}$ is a* non-extensible *inheritance if $R\mathcal{H}P$ implies the conditions in Definition 4.1, and also*

1. *if $R \xrightarrow{\tau} R'$ then $\exists P'.\ P \xrightarrow{\tau} P'$ and $R'\mathcal{H}P'$*
2. *if $R \xrightarrow{\bar{x}y} R'$ then $\exists P'.\ P \Longrightarrow\xrightarrow{\bar{x}y} P'$ and $R'\mathcal{H}P'$*
3. *if $R \xrightarrow{x(y)} R'$, $y \notin n(P,R)$ then $\exists P'.\ P \Longrightarrow\xrightarrow{x(y)} P'$ and $\forall w, R'\{w/y\}\mathcal{H}P'\{w/y\}$*
4. *if $R \xrightarrow{\bar{x}(y)} R'$, $y \notin n(P,R)$ then $\exists P'.\ P \Longrightarrow\xrightarrow{\bar{x}(y)} P'$ and $R'\mathcal{H}P'$*

Hence, the child agent R must not extend its parent P by offering new transitions, since any new action in R may synchronize with a complementary action in a certain $Q \in P\diamond$, where this transition was not considered when analyzing the compatibility of P and Q, since Definition 3.3 refers only to common complementary transitions. Thus, the definition of inheritance must be very restrictive, in order to preserve compatibility. However this restrictions will be overcome in the Definition 4.6 of role extension. Once again, non-extensible inheritance is a necessary condition for inheritance, but two additional conditions are required.

Definition 4.3 (Relation of inheritance) *A binary relation $\mathcal{H}$ on agents is an* inheritance *if $R\mathcal{H}P$ implies the conditions in Definitions 4.1 and 4.2, and also*

1. *if $P \equiv \mathbf{0}$ then $R \equiv \mathbf{0}$*
2. *if $P \stackrel{\tau}{\longrightarrow}$ then $\exists P', R'.\ P \stackrel{\tau}{\longrightarrow} P'$ and $R \Longrightarrow R'$ and $R'\mathcal{H}P'$*

The inheritance on agents $\dot{\rhd}$ is defined as the largest relation of inheritance.

The first condition states which agents inherit from the inaction. The second one refers to τ-transitions in the parent agent (which were not considered in Definition 4.1), and indicates that at least one of them must be inherited by the child. This condition maintains R synchronizable with any agent in $P\Diamond$, since actions complementary to local decisions in P are required for all of them. Consider again the agents $P = a(x).\mathbf{0} + \tau.b(y).\mathbf{0}$, and $R = a(x).\mathbf{0}$, which now fulfill all conditions for inheritance but that of Definition 4.3.2. Then we have that $R \dot{\diamond} Q$ for *most* $Q \in P\Diamond$, for instance $Q = \tau.\overline{a}(u).\mathbf{0} + \overline{b}(v).\mathbf{0}$, but for $Q' = \overline{b}(v).\mathbf{0}$ we have that $P \dot{\diamond} Q'$, but $R \not\dot{\diamond} Q'$. On the contrary, for $P' = \tau.a(x).\mathbf{0} + \tau.b(y).\mathbf{0}$ we have that $R\mathcal{H}P'$ (fulfilling now all the conditions for inheritance), $P' \dot{\diamond} Q$, and also that $R \dot{\diamond} Q$.

Definition 4.4 (Inheritance of behavior) R inherits from P*, written $R \rhd P$, if $R\sigma \dot{\rhd} P\sigma$ for all substitutions σ that respect the distinction $fn(P) \cup fn(R)$.*

The following result shows that inheritance preserves compatibility. An additional condition is necessary to prove the theorem, and for this reason, we reject *pathological* processes with infinite traces of local computations. This is not a restriction, since for these pathological roles will be typically of the form $P = \ldots + \tau.P + \ldots$, and we can always find a P' such that $P' \approx P$, without infinite traces of τ-transitions, where P' satisfies Definition 2.4.

Theorem 4.5 *Let P and Q be two agents, where P does not present any infinite trace of τ-transitions. Let $P \Diamond Q$. Let $R \rhd P$. Then, $R \Diamond Q$.*

Proof. It can be derived from Definitions 3.3 and 4.3. We only have to prove that $\Diamond_{inh} = \{(R, Q) : \exists P\ .\ P \Diamond Q \wedge R \rhd P\ \}$ is a relation of compatibility. □

Thus, inheritance preserves compatibility, and a single proof of inheritance ensures that a child role can be a substitute for any of its parents in any context, with no need to recheck compatibility. This result defines when a certain existing component can be used in an architecture; the roles of the component must inherit from those specified in the architecture. However, Definition 4.1 states that the child role cannot extend its parents, adding new behavior or functionality. We can overcome these restrictions defining *extension* as follows:

Definition 4.6 (Extension of behavior) R extends P*, written $R \!-\!\!\rhd P$, iff*

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

This definition relates extension to inheritance, and ensures successful attachment of the extended role R to any $Q \in P\Diamond$. In order to preserve compatibility, additional behavior in the child agent R is restricted, and it will not be used. However, this additional behavior may be successfully used in other architectures, even in combination with Q.

```
a) Client(request) = request‾(reply,error).
     ( reply(service).Client(request) + error.Client(request) )
b) Server(request) = request(reply,error).
     ( τ.reply‾(service).Server(request) + τ.error‾.Server(request) )
c) Client'(request) = request‾(reply,error).
     ( reply(service).Client'(request) + error.0)
d) FTServer(request) = (server)( FrontEnd(request,server) | Server(server) )
e) FrontEnd(request,server) =
     request(reply,error).FTService(request,server,reply)
   FTService(request,server,reply) = server‾(rep,err).
     ( rep(service).reply‾ service.FrontEnd(request,server)
     + err.FTService(request,server,reply) )
f) Server'(request) = request(reply,error).reply‾(service).Server'(request)
```

Figure 2: Client/Server System

Example 2. A typical example of SA is that of Client/Server systems. Suppose that the client requests a service, and either obtains it or gets an error, as indicates role `Client` in Figure 2 (a). On the other hand, suppose that the server, represented by the role in Figure 2 (b), may fail to serve some of the requests (local choices are used to represent this internal decision). Using Definition 3.3 it is trivial to find out that `Client` $\diamond$ `Server`. But suppose now that we have a component which behaves as described in role `Client'` (c), crashing when an error is received. This component is not compatible with our `Server`, since the synchronization of both roles in the event `error` will lead `Client'` to inaction while the server will be deadlocked in state `Server` (in absence of more clients).

However, we can develop a fault-tolerant server `FTServer` (d), wrapping our `Server` (b) with a component `FrontEnd` (e) which collects requests from the client and retransmits them to the `Server` until the service is obtained. From the point of view of the `Server` (b), `FrontEnd` behaves like a client, and `Client(server)` is a correct role of `FrontEnd(request,server)`, hiding link `request`. Now, from `Client` $\diamond$ `Server`, we can derive that the architecture of `FTServer` is safe, and this component can be built by combining its subcomponents as described in (d).

On the other hand, it can be proved that role `Server'` (f) is a correct role of `FTServer`, and also that `Client'` $\diamond$ `Server'`. Hence, `FTServer` can collaborate with clients that behave like `Client'`.

In addition, it can be proved using Definition 4.3 that `Server'` $\triangleright$ `Server`. Hence, from Definition 4.5, we have that `Server'` is also compatible with `Client` (a), and we can claim that `FTServer` is a derived version of `Server` which conforms the requirements of our Client/Server architecture.

Although the example is very simple, it shows how compatibility and inheritance can be applied to the incremental development of more complex systems.

5 Discussion and Related Work

The specification and analysis of software architectures becomes an important task when dealing with complex systems. In this work, we have shown how the use of π-calculus may address the needs of software architects. Mobility is easily specified in π-calculus, which makes our proposal specially interesting for dynamic architectures. We have given a definition of role in the context of π-calculus, and described how roles can be derived from the specification of a component.

Since bisimilarity is not adequate for comparing roles, we have defined a relation of compatibility which formalizes the notion of conformance of behavior among components. We have also defined relations of role inheritance and extension. These relations permit the replacement of components with specialized versions, maintaining the compatibility of the system with no need of checking the attachments modified by the replacement.

There are several interesting approaches to protocol conformance in π-calculus. First of all, Milner's *sorting* [Mil92] introduces a notion of types for the polyadic π-calculus which has been extensively studied afterwards. Another approach is that of [Yos96], which using graph types extends sorting with dynamic aspects of process behavior, giving a definition of *protocol type*, as a pair of complementary graph types. However, our work differs from theirs in that our goal is not to define protocol equivalence or complementariety, but to check compatibility and refinement of behavior as a way of determining the conformance of a component to the requirements imposed by a certain system or architecture. Thus, the relations of compatibility and inheritance presented in this work allow a safe and flexible way of connecting software components even though their behavior is not strictly the "same".

In the last years, SA has deserved active research interest [GP95, MK96]. Although most of proposals in this field are not formally founded, several papers have already proposed the use of formalisms such as CSP or CHAM for architecture specification. In [MK96] the π-calculus is used for defining the semantics of Darwin. However, type checking is reduced in Darwin to name equivalence, and aspects like compatibility and inheritance of behavior are not considered.

Our definition of compatibility follows the ideas developed in [AG97], which uses CSP to specify software components using asymmetric ports and roles. However, CSP does not seem appropriate for the description of structures with changing communication topology. At most, CSP can be used in systems with a finite number of configurations, as they show in [ADG98]. On the contrary, our approach is able to deal with dynamic architectures, in which components may enter and leave the system at run-time, and which present an evolving communication topology. Furthermore, [AG97] does not address issues of inheritance or extension, nor it suggests a methodology for role description.

Nevertheless, π-calculus is a low-level notation, which makes difficult its application to industrial-size systems. Hence, we are developing LEDA [CPT99a], a higher-level ADL which is founded on the π-calculus. LEDA integrates the relations of compatibility and inheritance presented in this work, and will serve to evaluate their possibilities to solve real problems in software development. A case study showing the application of our approach to a more realistic example than those presented here can be found in [CPT99b].

A The π-calculus

The π-calculus is a process algebra specially suited for the description and analysis of concurrent systems with dynamic or evolving topology. Systems are specified in the π-calculus as collections of processes or *agents* which interact by means of links or *names*. The calculus allows direct expression of mobility, which is achieved by passing link names as arguments or *objects* of messages. When an agent receives a name, it can use this name as a *subject* for future transmissions, which allows an effective reconfiguration of the system. In fact, the calculus does not distinguish between links and data. This homogeneous treatment of names is used to construct a very simple but powerful calculus.

Let $P, Q, \ldots$ range over agents and $w, x, y, \ldots$ range over names. Sequences of names are abbreviated using tildes ($\tilde{w}$). Then, agents are built as follows:

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

Inactive behavior is represented by the inaction $\mathbf{0}$. Restrictions are used to create private names. Thus, in *(x)P*, x is private to P, and it can only be used as a subject for communication within P. However, the scope of a name is widened simply by sending it to another agent. A match *[x=z]P* behaves like P if x and z are identical, otherwise like $\mathbf{0}$. Though matching is unnecessary, and the control it provides can be achieved by other means, it is often used for obtaining simpler encodings.

Silent transitions, given by τ, model internal actions. Thus, an agent $\tau.P$ will eventually evolve to P. An output-prefixed agent $\bar{x}y$.P sends the name y (object) along name x (subject) and then continues like P. An input-prefixed agent *x(w).P* waits for a name y to be sent along x and then behaves like $P\{y/w\}$, where $\{y/w\}$ is the substitution of w with y. Apart from these three basic actions, there is also a derived one –*bound output*, expressed $\bar{x}(y)$–, which represents the emission of a *private* name y, widening the scope of this name. Bound output is just a short form for $(y)\bar{x}y$, but it must be considered separately since it has different transition rules than free output actions.

In the monadic π-calculus, only one name at a time can be used as object in an input or output action. However, a polyadic version, allowing the communication of several names in one single action, can be found in [Mil92]. Using the so called *molecular actions*, it is trivial to translate any polyadic encoding into monadic.

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

$$(\cdots + \bar{x}y.P + \cdots) \mid (\cdots + x(w).Q + \cdots) \xrightarrow{\tau} P \mid Q\{y/w\}$$

On the other hand, local choices are expressed combining the summation operator with silent actions. Hence, an agent like $(\cdots + \tau.P_1 + \tau.P_2 + \cdots)$ may proceed to P_1 or P_2 with independence of its context.

Both the composition and summation operators can be applied to a finite set of agents $\{P_i\}_i$. In this case, they are represented as $\prod_i P_i$ and $\sum_i P_i$, respectively.

Finally, $A(\tilde{w})$ is a defined agent. Each agent identifier A is defined by an equation $A(\tilde{w}) = P$. Agent identifiers allows recursive definition of agents.

The names in an agent P are denoted by $n(P)$. The *free names* of P, $fn(P)$, are those names in P not bound by an input action or a restriction. The *bound names* of P are denoted by $bn(P)$.

Substitutions, represented by σ, are defined in the expected way. On the other hand, *distinctions*, defined as sets of names, forbid the identification of certain names. Thus, a substitution respects a distinction if it does not equate any two names in the set.

Structural congruence for the π-calculus, denoted by $\equiv$, is defined in several papers, in particular in [Mil92]. On the other hand, several relations of equivalence have been proposed for this calculus. In this paper we refer to Milner's strong and weak bisimilarity, respectively $\sim$ and $\approx$.

Constants are considered as names in the π-calculus, with the particularity that they are never instanced. In order to avoid confusion with other names in the specification they are written with a different font (CONSTANT). For simplicity, they are not included among the free names of any agent identifier that uses them.

Finally, the following shorthand is commonly accepted:

- $A \xrightarrow{\alpha}$ stands for $\exists A' \,.\, A \xrightarrow{\alpha} A'$.

- $\Longrightarrow$ stands for $(\stackrel{\tau}{\longrightarrow})^*$, the reflexive and transitive closure of $\stackrel{\tau}{\longrightarrow}$.
- $\stackrel{\alpha}{\Longrightarrow}$ stands for $\Longrightarrow\stackrel{\alpha}{\longrightarrow}\Longrightarrow$ when $\alpha \neq \tau$.

For a detailed description of the calculus, including its transition system, we refer to [MPW92].

References

[ADG98] R. Allen, R. Doucence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proc. ETAPS'98*, Lisbon, 1998.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, July 1997.

[Bou87] G. Boudol. Notes on algebraic calculi of processes. In *Logics and Models of Concurrent Systems*, number 13 in NATO ASI series. K.Apt, 1987.

[CPT97] C. Canal, E. Pimentel, and J.M. Troya. On the composition and extension of software systems. In *Proc. FSE'97 FoCBS Workshop*, pages 50–59, Zurich, September 1997.

[CPT98] C. Canal, E. Pimentel, and J.M. Troya. Compatibility, inheritance and extension of π-calculus agents. Technical Report LCC-ITI-98-13, Universidad de Málaga, June 1998. http://www.lcc.uma.es/˜canal/LCC-ITI-98-13.

[CPT99a] C. Canal, E. Pimentel, and J.M. Troya. Specification and refinement of dynamic software architectures. In P. Donohoe, editor, *Software Architecture*, pages 107–126. Kluwer Academic Publishers, 1999.

[CPT99b] C. Canal, E. Pimentel, and J.M. Troya. Specification of interacting software components: a case study. In *Proc. Ideas'99*, pages 381–392, San José (Costa Rica), March 1999.

[GP95] D. Garlan and D.E. Perry. Special issue on software architecture. *IEEE Trans. on Software Engineering*, 21(4), April 1995.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil92] R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification.* Springer, 1992.

[MK96] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proc. ACM FSE'96*, pages 3–14, San Francisco, October 1996.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.

[NM95] O. Nierstrasz and T.D. Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[SG96] M. Shaw and D. Garlan. *Software Architecture.* Prentice Hall, 1996.

[Yos96] N. Yoshida. Graph types for monadic mobile processes. In *Proc. FST TCS'96*, number 1180 in LNCS, pages 371–386. Springer Verlag, 1996.