Compatibility and Inheritance in Software Architectures

Carlos Canal Ernesto Pimentel José M. Troya

Depto. de Lenguajes y Ciencias de la Computación,
Universidad de Málaga
Campus de Teatinos, 29071 Málaga, Spain
{*canal,ernesto,troya*}*@lcc.uma.es*

Abstract

The application of formal methods to the development of software depends on the availability of adequate models and formalisms for each of the stages of the development process. In this work, we focus on the level of design called Software Architecture. At this level, the system is described as a collection of interrelated components, and it is here where the properties derived from system's structure can be naturally analyzed. Our approach uses process algebras as a formal basis for the description and analysis of software architectures. Process algebras are widely accepted for the specification of software systems. In particular π-calculus addresses the description of systems with a dynamic or evolving topology, and permits their analysis for bisimilarity and other interesting properties. Though bisimilarity determines the equivalence of behavior, more flexible relations are needed in the context of Software Engineering, in order to support formally the notions of conformance and refinement of behavior. 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 can be considered as a specialization or extension of another one. The suitability of our approach is shown by its application to the field of Software Architecture [1].

Keywords: theory of concurrency, process calculi, π-calculus, software architecture, compatibility and inheritance of behavior

[1] This work is sponsored by the Spanish Government (*Comisión Interministerial de Ciencia y Tecnología*) under contracts TIC98-445-C03-03 and TIC99-1083-C02-01

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, which have been successfully applied in a number of different and heterogeneous contexts. Although the calculus has evolved from its original definition [1], incorporating several extensions (e.g. higher order, asynchronous communication, etc.), the basic ideas have remained without substantial changes.

Unlike other process calculi, such as CCS or CSP, the π-calculus can express *mobility* in a direct manner by allowing references to processes, or *links*, to be passed as values in communication. This makes the π-calculus specially suited for describing the structure of software systems in which components can be dynamically created and removed, and in which attachments between components are also dynamically established and modified, leading to an evolving communication topology. We say that these systems present a *dynamic architecture.* Typical examples of this kind of applications are open and distributed software systems. The specific characteristics of these large and dynamic systems require to make changes in the methods and tools of Software Engineering in order to cope with the new requirements. In this context, the applicability of the ideas and results presented in this paper is shown in [2], where we propose the extension of standard component interfaces with protocol information, specified using the π-calculus, and their analysis by means of the formal underpinnings presented here.

The advantage of using an algebraic calculus to specify concurrent systems is the capability to make some kind of analysis on the expected behavior of the system. In this way, some properties of safety and liveness can be verified when the system is specified in terms of a set of interacting agents. Moreover, the corresponding analysis can be automated [3]. Thus, it is usual to obtain information about the equivalence of two processes, or about situations of deadlock. The latter is specially relevant in the specification of concurrent systems. In fact, one of the most common mistakes when programming or specifying this kind of systems is the presence of behavioral mismatches leading to undesirable deadlocks. On the other hand, processes in π-calculus can be compared in order to know whether they are "equivalent", that is, whether their behavior coincides or not. This information can be used either to substitute parts in a system by equivalent ones, or to justify a certain strategy for program transformation. In any case, the fundamental notion which is behind of these ideas is that of (weak or strong) *bisimilarity.* This relationship defines when two processes present a "similar" behavior.

However, some interesting properties which characterize the components (i.e. parts) of a system are not directly related with bisimilarity. On the contrary, we can find situations where other ways for comparing processes are more convenient from the software engineer's point of view. In this sense, one of the properties that we may analyze is whether the components of the system conform to each other or not. 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.

That is the situation in several fields of Software Engineering currently deserving active research, such as Software Architecture, Component-Orientation, or Framework-based software development. Following these approaches, software systems are structured as a collection of interacting computational and data components [4], focusing on those aspects of design and development which cannot be suitably treated inside the modules which compose the system [5]. 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. Similar studies have been made by Allen and Garlan [6] in the context of CSP, although their results cannot be easily extended to π-calculus due to the characteristics of mobility present in this calculus.

Compatibility could be determined by global analysis of the system. However, this is impractical for complex systems. Instead of that, we use partial interface specifications or *roles* to describe the behavior of each component. Thus, the system is described by a set of role-to-role attachments, representing the interconnection of the corresponding components, and each pair of attached roles is locally checked for compatibility [7]. This reduces the complexity of the analysis, and justifies the use of the compatibility relation as an analysis tool.

On the other hand, strong bisimilarity is a congruence in the context of the π-calculus, which supports the replacement of processes, guaranteeing that the global behavior of the system is not affected. However, effective reuse of a software component often requires that some of its parts can be removed, reconfigured or specialized to accommodate them to new requirements [8]. Again, current works on process algebras, in general, and on π-calculus in particular, do not deal with this kind of problem. In this way, our approach is completed with the definition of new relations of inheritance and extension for processes in the context of π-calculus. These relations preserve compatibility, allowing the specification of polymorphic behaviors, and promoting both incremental specification and reusability.

The relations of compatibility and inheritance of behavior presented in this paper have been applied in the development of LEDA [9], an Architecture Description Language (ADL) based on the π-calculus. However, it should be noticed that these relations are applicable not only to the context of Software Architecture, but also to the analysis of processes in general.

The rest of this work is structured as follows. First, we present a short introduction to the π-calculus. In Section 3 we propose the use of the π-calculus for the specification and validation of software architectures, and we formalize the notions of role, attachment and architecture in the context of this calculus, giving also a methodology for the derivation of roles from components. Then, Section 4 contains the definition of a relation of behavioral conformance or compatibility, and presents some interesting results on how role compatibility ensures successful composition of the corresponding components. Next, Section 5 defines compatibility-preserving relations of inheritance and extension among processes. We conclude discussing the originality and relevance of our approach, comparing it with some related works.

2 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*. These names can be considered as shared bidirectional channels, which act as *subjects* for communication. Scope restriction allows to establish links that are private to a group of agents. The π-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 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.

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

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

Empty or inactive behavior is represented by the inaction $\mathbf{0}$. Restrictions are used to create private names. Thus, in $(x)P$, the name x is private to P.

Communication using x as subject is prohibited between P and any other agent, but it is allowed inside P, i.e. between its components. The scope of a name can be widened simply by sending it to another agent (see *bound output* below). A match $[x = z]P$ behaves like P if the names x and z are identical, and otherwise like $\mathbf{0}$. Though matching is unnecessary for computations over data types, which can be achieved by other means, we use it in order to obtain easier encodings.

Silent transitions, given by τ, model internal actions. Thus, an agent $\tau.P$ will eventually evolve to P without interacting with its environment. 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 transitions, there is also a derived one –*bound output*, expressed $\bar{x}(y)$–, which represents the emission along a link x 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 slightly 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 [10]. Using the so called *molecular actions*, it is trivial to translate any polyadic encoding into monadic. In this work we use polyadic encodings when required.

The composition operator is defined in the expected way: $P \mid Q$ consists of P and Q acting in parallel. The summation operator is used for specifying alternatives: $P + Q$ may proceed to P or Q. The choice may be locally or globally taken. In a global choice, two agents agree synchronously 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 $(\cdots + \tau.P + \tau.Q + \cdots)$ may proceed to P or Q with independence of its context. We use local and global choices to state the responsibilities for action and reaction.

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 a unique equation $A(\tilde{w}) = P$. The use of agent identifiers allows modular and recursive definition of agents.

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

Structural congruence for the π-calculus, is defined in several papers, in particular in [10].

Definition 2.1 *Structural congruence, denoted by $\equiv$, is the smallest congruence relation over $\mathcal{P}$ such that*

- $P \equiv Q$ *if they only differ by a change of bound names.*
- $(\mathcal{N}/\equiv, +, \mathbf{0})$ *is a symmetric monoid.*
- $(\mathcal{P}/\equiv, \mid, \mathbf{0})$ *is a symmetric monoid.*
- $(x)\mathbf{0} \equiv \mathbf{0}$*, and* $(x)(y)P \equiv (y)(x)P$.
- *If* $x \notin fn(P)$ *then* $(x)(P \mid Q) \equiv P \mid (x)Q$.

Transitions are represented by labeled arrows. Hence, $P \xrightarrow{\alpha} P'$ indicates that the process P performs an action α and then becomes P'. Apart from this basic transition, we use the following shorthand along this paper:

- $\Longrightarrow$ stands for $(\xrightarrow{\tau})^*$, the reflexive and transitive closure of $\xrightarrow{\tau}$.
- $\overset{\alpha}{\Longrightarrow}$ stands for $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$ when $\alpha \neq \tau$.
- $P \xrightarrow{\alpha}$ stands for $\exists\, P' \,.\, P \xrightarrow{\alpha} P'$.
- $P \Longrightarrow \mathbf{0}$ stands for $\exists\ \mathbf{0}_P \,.\, P \Longrightarrow \mathbf{0}_P$ and $\mathbf{0}_P \equiv \mathbf{0}$.

The transition system that we are considering is that proposed in [1], and it is shown in Figure 1. The transition system is closed with respect to structural congruence.

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 bind any two names in the set. As it will be shown, we use distinctions to avoid conflicts among free names in an agent.

Constants are considered as names in the π-calculus, with the particularity that they are never instantiated. In order to avoid confusion with other names in the specification we write them in small capitals (Constant). For simplicity, they are not included among the free names of any agent identifier that uses them.

Several relations of equivalence have been proposed for this calculus. In this paper we refer to Milner's strong and weak bisimilarity, respectively denoted by $\sim$ and $\approx$.

$$\text{TAU: } \frac{-}{\tau.P \xrightarrow{\tau} P} \qquad \text{OUT: } \frac{-}{\bar{x}y.P \xrightarrow{\bar{x}y} P} \qquad \text{IN: } \frac{-}{x(z).P \xrightarrow{x(w)} P\{w/z\}}\ w \notin fn((z)P)$$

$$\text{MATCH: } \frac{P \xrightarrow{\alpha} P'}{[x=x]P \xrightarrow{\alpha} P'} \qquad \text{IDE: } \frac{P\{\tilde{y}/\tilde{x}\} \xrightarrow{\alpha} P'}{A(\tilde{y}) \xrightarrow{\alpha} P'}\ A(\tilde{x}) =_{def} P$$

$$\text{SUM: } \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \text{PAR: } \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ bn(\alpha) \cap fn(Q) = \varnothing$$

$$\text{COM: } \frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}} \qquad \text{CLOSE: } \frac{P \xrightarrow{\bar{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P \mid Q \xrightarrow{\tau} (w)(P' \mid Q')}$$

$$\text{RES: } \frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'}\ y \notin n(\alpha) \qquad \text{OPEN: } \frac{P \xrightarrow{\bar{x}y} P'}{(y)P \xrightarrow{\bar{x}(w)} P'\{w/y\}}\ y \neq x \wedge w \notin fn((y)P')$$

Fig. 1. Transition System for π-calculus

Definition 2.2 *$\mathcal{S}$ is a simulation if $P\mathcal{S}Q$ implies that*

(1) If $P \xrightarrow{\tau} P'$, then for some Q', $Q \xrightarrow{\tau} Q'$ and $P'\mathcal{S}Q'$.
(2) If $P \xrightarrow{\bar{x}y} P'$, then for some Q', $Q \xrightarrow{\bar{x}y} Q'$ and $P'\mathcal{S}Q'$.
(3) If $P \xrightarrow{x(y)} P'$ and $y \notin n(P,Q)$,
then for some Q', $Q \xrightarrow{x(y)} Q'$ and $\forall\, w\ P'\{w/y\}\mathcal{S}P'\{w/y\}$.
(4) If $P \xrightarrow{\bar{x}(y)} P'$,and $y \notin n(P,Q)$,
then for some Q', $Q \xrightarrow{\bar{x}(y)} Q'$ and $P'\mathcal{S}Q'$.

A binary relation $\mathcal{S}$ is a *bisimulation* if both $\mathcal{S}$ and its reverse are simulations. *Ground bisimilarity*, written $\dot{\sim}$, is defined as the largest bisimulation. Finally, P and Q are strongly bisimilar, written $P \sim Q$, if $P\sigma \mathrel{\dot{\sim}} Q\sigma$ for all substitutions σ. The weak version, written $\approx$, is obtained ignoring silent τ actions by replacing the arrows $\xrightarrow{\alpha}$ from Q with $\overset{\alpha}{\Longrightarrow}$ if $\alpha \neq \tau$, or with $\Longrightarrow$ if $\alpha = \tau$.

The definition of bisimilarity involves universal quantification over substitutions, which increases largely the size of the relations needed to define a bisimulation. However, it is possible to develop a more efficient transition system, as it is done [11], and also automatizable algorithms for the relations of bisimilarity [12], which allows the development of analysis tools.

Some examples of agents written in π-calculus can be found in the following sections, but for a detailed description of the calculus we refer to [1].

3 Specification of Software Architectures

As software technology becomes a core part of business corporations in all market sectors, customers demand more flexible systems. In this context, the increasing use of personal computers and easy access to local and global communication networks provide an excellent infrastructure for building open distributed systems. However, the specific problems of those large and dynamic systems are currently challenging the Software Engineering community, whose traditional methods and tools are finding difficulties for coping with the new requirements.

In the last few years, the term Software Architecture (SA) has been adopted for referring to the level of software design in which the system is represented as a collection of computational and data components interconnected in a certain way [4]. SA focuses in those properties of software systems which derive from their structure, i.e. from the way in which their components are combined. The importance of explicit architectural specifications is widely accepted. First, they raise the level of abstraction, making easier the comprehension of complex systems. Second, they promote reuse of both architectures and components.

We propose the use of the π-calculus for the specification of software architectures. Apart from the opportunities for formal analysis that are present in process algebras, direct expression of mobility in the π-calculus allows the description of architectures with changing communication topology, what can be hardly done using other formalisms like CSP or CCS. In fact, the kind of dynamic reconfiguration present in open and distributed software systems is expressed very naturally in the π-calculus by passing link names among agents in the same way that references to components, *sockets* or URL addresses are interchanged among software components in a dynamic and distributed system.

It should be noticed that although we always represent software composition by means of the parallel operator ($|$) of the π-calculus, our approach is not restricted only to the specification and validation of architectures consisting just of a concurrent composition of components. On the contrary, parallel composition is the natural representation in process algebras of any form of software composition. Hence, even sequential systems can be naturally represented as a parallel composition of its components.

Software systems can be described in π-calculus by composing the specifications of their components. Connections among components will be represented by shared names in the corresponding component specifications. 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, and it will be hidden by the details of components' specifications. Second, state explosion would prevent the analysis of the specifications of complex systems.

Instead of that, we use partial interface specifications, or *roles*, for expressing the behavior of each component, and describe explicitly system architecture as a set of attachments between roles, representing the interconnection of the corresponding components. Behavioral conformance or mismatch among system components will be determined by analyzing the compatibility of the roles that represent them, reducing the complexity of the analysis to a great extent. This kind of local analysis cannot ensure global properties such that the whole system is deadlock-free, but this is not our goal. As it will be shown, local analysis of compatibility guarantees deadlock-freedom while combining a component with the roles specified in a given architecture, ensuring that the component "matches" the architecture, and that there is no behavioral mismatch between component interfaces, which stands for system composability. Attachments between compatible roles are able to interact successfully, indicating full conformance of the corresponding components. On the other hand, a mismatch detected when analyzing the compatibility of an attachment among roles stands for a mismatch in the behavior of the corresponding components, which will lead to a system crash or failure.

3.1 Roles, attachments and architectures

Before giving a formal definition for the notion of role, some issues must be addressed. First of all, roles will be specified by agents on the π-calculus. Hence, we must be able to distinguish between successful termination and failure in the context of this process algebra. However, in the π-calculus both process which *are* the inaction, like $\mathbf{0} \mid \mathbf{0}$, and those which *behave* like the inaction, like $(a, b)(a(w).\mathbf{0} \mid \bar{b}y.\mathbf{0})$ are strongly bisimilar, so bisimulation does not provide a way to differentiate them.

Termination, deadlock, and divergence in process algebras has been addressed in several works, particularly in [13]. Following a similar approach, we introduce here a definition of success and failure in the context of the π-calculus in which structural congruence plays a significant role.

Definition 3.1 (Success and Failure) *An agent P is a* failure *if exists P' such that $P \Longrightarrow P'$, $P' \stackrel{\tau}{\not\longrightarrow}$, and $P' \not\equiv \mathbf{0}$.*

On the contrary, an agent P is successful *if it is not a failure.*

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

The definition of role below is based on the notion of *abstraction* in process algebras, as first established in [14]. Ours is an adaptation to the π-calculus based on a *Hiding* operator ($/_{\tilde{x}}$) similar to that defined in [15] for CCS, but taking into account the characteristics of mobility present in the π-calculus:

Definition 3.2 (Hiding) *Let P be an agent and $\tilde{x} \subseteq fn(P)$, then*

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

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

For each link name $x \in \tilde{x}$ the agent $Ever(x)$ hides x in P. $Ever(x)$ is always willing to perform input and output actions along x (and also along any other link name sent or received through x). Thus, it satisfies the need of communication of P along these links and avoids deadlocks. Hence, the actions in P along link x are shown in the agent $P/_x$, which hides this name, as silent actions τ.

Now, we can define roles as follows:

Definition 3.3 (Role) *An agent P is a role of a component Comp if*

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

Notice that in the definition above, roles are a semantic notion, defined up to weak bisimilarity. Thus, given a component *Comp* and a subset of its free names, several roles can be derived. However, not all of them will be meaningful abstractions of the interface of *Comp*. Furthermore, some may not be even correct abstractions, and a certain method for deriving roles from components is required. We will address these topics in Definition 3.5 and Section 3.2 below.

Definition 3.3 states that the free names in a role are a subset of those of the corresponding component. Hence, roles are obtained from components by *hiding* the names which are not relevant to the partial interface represented by the role, and a role only contains a subset of the non-silent transitions of the corresponding component. Any component action using hidden names will appear in the role as a silent transition. When these actions are combined in the component with the sum operator, they will appear as local choices in the role, since from the point of view of a component connected to this role, we can't say if these transitions will take place or not.

Example 3.4 *Consider a Translator, a very simple component which copies the data received in its input links i_1 and i_2 to its output links o_1 and o_2, hence* *performing a sort of translation between input and output link names.*

$$\begin{aligned} Translator(i_1, i_2, o_1, o_2) &= i_1(x).\bar{o}_1 x.Translator(i_1, i_2, o_1, o_2) \\ &+ i_2(y).\bar{o}_2 y.Translator(i_1, i_2, o_1, o_2) \end{aligned}$$

The interface of this component can be divided into two roles, Input and Output, with free names i_1, i_2 and o_1, o_2 respectively. One specification of these roles satisfying Definition 3.3 is as follows:

$$Input(i_1, i_2) = i_1(x).\tau.Input(i_1, i_2) + i_2(y).\tau.Input(i_1, i_2)$$

$$Output(o_1, o_2) = \tau.\bar{o}_1(x).Output(o_1, o_2) + \tau.\bar{o}_2(y).Output(o_1, o_2)$$

If we observe the Translator from its output role, we can't say in advance which output action will be performed, since it depends on the previous input action, which is part of a different role. This internal decision is modeled in the role above as a local choice by combining τ-actions with the sum operator.

Notice that output actions through links o_1 and o_2 are free *in component Translator while the same actions are* bound *in its role Output. This transformation of free output actions to bound ones is subtle, but it occurs very often in roles, so it deserves some explanation. The names x and y used as objects in the output actions of the Translator were obtained as a result of a previous input action on links i_1 and i_2, respectively. Hence, these output actions are free in the component.*

However, in the role Output, links i_1 and i_2 are hidden, and input actions through these names simply appear as τ-actions. Therefore, the names in the output actions of the role must be considered as fresh, and these actions are now bound. This transformation, which satisfies Definition 3.3, is also meaningful. From the point of view of a component connected to Translator's role Output, the names x and y received from this component are new, and cannot be traced to its origin in the hidden input actions through links i_1 and i_2.

As we have stated before, roles are defined up to weak bisimilarity, but not every agent which satisfies the conditions in the definition of roles can be considered as a *correct* role for a given component. For instance, consider the component $Comp(a, b) = a(x).Comp(a, b) + b(y).\mathbf{0}$, and two agents which satisfy Definition 3.3, $P_1(a) = a(x).P_1(a) + \tau.\mathbf{0}$, and $P_2(b) = b(y).(c)c.\mathbf{0}$. While P_1 is a correct role for $Comp$, P_2 is not, since action $b(z)$ leads $Comp$ to success, and P_2 to failure. The aim of the definition below is to establish the correction of roles.

Definition 3.5 (Correctness) *Let Comp be a component and* $\mathcal{P} = \{P_1 \mathinner{..} P_n\}$ *a set of its roles. Comp is correctly specified by* $\mathcal{P}$ *iff*

(1) $\mathit{fn}(P_i) \cap \mathit{fn}(P_j) = \varnothing \quad \forall\, i \neq j$
(2) $\mathit{Comp} \Longrightarrow \mathbf{0}$ *iff* $\forall\, i\ P_i \Longrightarrow \mathbf{0}$
(3) If $\exists\, \alpha\ (\alpha \neq \tau)\ .\ \mathit{Comp} \overset{\alpha}{\Longrightarrow} \mathit{Comp}'$*, then* $\exists\, i, P_i'\ .\ P_i \overset{\alpha}{\Longrightarrow} P_i'$*, and* Comp' *is correctly specified by* $\mathcal{P}' = \{P_1, \ldots, P_i', \ldots, P_n\}$

Definition 3.5 indicates that a component will be specified by a set of roles, each of them referring to the interaction with another component. These roles are disjoint abstractions partially describing the behavior of the component as seen from its environment, and abstracting behaviors which are not relevant to the role. Furthermore, the interface of the component *Comp* must be completely specified by the roles in $\mathcal{P}$, and the sets of free names in the roles must be disjoint. The latter ensures that the specification of the component is made modularly, in such a way that different roles cannot synchronize. Thus, when $P_1 \mid \cdots \mid P_i \mid \cdots \mid P_n \xrightarrow{\tau} P$ some P_i performs a τ-transition to P_i' and $P = P_1 \mid \cdots \mid P_i' \mid \cdots \mid P_n$. Similarly, when $P_1 \mid \cdots \mid P_i \mid \cdots \mid P_n \xrightarrow{\alpha} P$ $(\alpha \neq \tau)$, exactly *one* P_i has a transition α to P_i' and $P = P_1 \mid \cdots \mid P_i' \mid \cdots \mid P_n$. Definitions 3.3 and 3.5 make possible the derivation of roles from the specification of the corresponding components.

Example 3.6 *Consider a Buffer which provides common put and get operations.*

$$\mathit{Buffer}(\mathit{put}, \mathit{get}) = \mathit{BufferSpec}(\mathit{put}, \mathit{get}, \textsc{Null}, \mathrm{T})$$

$$\begin{aligned} &\mathit{BufferSpec}(\mathit{put}, \mathit{get}, \mathit{node}, \mathit{empty}) = \\ &\quad \mathit{put}(\mathit{it}).(n)(\mathit{Node}(n, \mathit{it}, \mathit{node}, \mathit{empty}) \mid \mathit{BufferSpec}(\mathit{put}, \mathit{get}, n, \mathrm{F})\ ) \\ &\ + [\mathit{empty} = \mathrm{F}]\mathit{get}.\mathit{node}(\mathit{it}, \mathit{next}, \mathit{last}).\overline{\mathit{get}}\ \mathit{it}.\mathit{BufferSpec}(\mathit{put}, \mathit{get}, \mathit{next}, \mathit{last}) \end{aligned}$$

$$\mathit{Node}(\mathit{node}, \mathit{it}, \mathit{next}, \mathit{last}) = \overline{\mathit{node}}\ \mathit{it}\ \mathit{next}\ \mathit{last}.\mathbf{0}$$

This component can play two different roles in a system: data storage and data retrieval. Then, the interface of the Buffer is divided into roles – Storage and Retrieval –, in which actions referring to hidden names would be represented by τ*-actions. Since these internal actions do not stand for local choices they are omitted in the roles, which specify correctly and completely the Buffer.*

$$\mathit{Storage}(\mathit{put}) = \mathit{put}(\mathit{it}).\mathit{Storage}(\mathit{put})$$

$$\mathit{Retrieval}(\mathit{get}) = \mathit{get}.\overline{\mathit{get}}(\mathit{it}).\mathit{Retrieval}(\mathit{get})$$

In a typical Producer/Consumer system, these roles will 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 Producer/Consumer system, with no need of reasoning about the complete specification of the buffer, the producer, or the consumer.*

The connection of several components in a certain architecture will be represented by an attachment among roles of these components. In order to avoid synchronization between different attachments, the free names of the roles must be conveniently restricted. Hence, attachments are defined as follows.

Definition 3.7 (Attachment) *Let $\{P_i\}_i$ be a set of roles. Their attachment is defined as*

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

Finally, we can define an *architecture*, formed by the composition of several components, as a set of attachments between roles of these components.

Definition 3.8 (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 each $Comp_j$ $(j = 1 \,..\, n)$. 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 some of the results of the following sections, we will first consider *binary* architectures, where each $\mathcal{R}oles_k$ attaches only a pair of roles. Then, these results will be extended to general architectures.

3.2 Obtaining roles from components

Definitions 3.3 and 3.5 establish the conditions for finding out if a group of agents are correct roles for a given component. However, these definitions do not identify only one suite of roles for representing a component. Hence, a sort of methodology is required. Some of the basic ideas of this methodology are scattered through all this work, here we try to organize them, giving useful hints for obtaining roles from components.

The first issue that may raise refers to how many roles a given component should have. There is no definite answer to this question. If we choose to specify the whole interface of the component using one single role, we wouldn't achieve much abstraction, and only some internal details of the component could be hidden in the role, resulting in a complex role, difficult to analyse.

If we choose to specify one role for each free name in the interface of the component, probably we will break the component into too many separate pieces, each of them giving a vision of its behavior too narrow to be meaningful and useful. As a general rule, the correct answer is to divide the interface of the component into as many roles as components are connected to that one under consideration. Then, each role will refer to the interaction between two components –the one being described and another one which is connected to it– and will represent one of the *rôles* that the component plays in the system. Each role will hide all the free names in the component but those used for communication with the component connected to it. Following this rule, most of the components described in the examples shown in this work are represented by two roles, since they are connected to two other components (see for instance Examples 3.4 and 3.6).

However, Definition 3.5 indicates that a set of correct roles must make a disjoint partition of the interface of a component. Hence, if several components interact with a certain one through a common name or names, we will probably have a multiple (not binary) attachment among roles of all these components.

Once we have decided which are the roles of a component (and consequently which are the free names considered in each role), roles are obtained by hiding in the component all link names but these. Input and output actions involving hidden names are transformed into τ-actions in the role, causing a local choice when combined with the sum operator, as shown in Example 3.4 for role *Output*. Names sent or received in the so hidden actions, are also hidden. On the contrary, actions through names which are still free in the role, remain the same (provided some free output actions maybe transformed into bound outputs, as shown in the already mentioned role *Output*). Finally, a recursive definition in a component causes also recursion in the corresponding role.

In the same way as input and output actions in a component may appear as local choices in the corresponding role when these actions are through hidden link names, also component actions constrained by match operators on hidden names should appear as local choices in the role. Once again, the reason is that, from the point of view of the environment, the commitment to these transitions will be locally decided. This transformation of match operations into silent transitions is done in two steps. As stated in [1], match operations over data types are unnecessary in the π-calculus, and they can be replaced by actions among the names involved. Then, as these actions use hidden names, they are abstracted by silent transitions in the role.

Example 3.9 *Consider an Observer, a component which receives colored balls from a BallGenerator and sends "red" or "black" events through its output port depending on the color of the ball. These components can be specified using matching as follows,*

$$\begin{aligned}
&\mathit{Observer}(\mathit{ball}, \mathit{red}, \mathit{black}) = \mathit{ball}(\mathit{color}).\\
&\quad (\ \ [\mathit{color} = \textsc{Red}]\ \overline{\mathit{red}}.\mathit{Observer}(\mathit{ball}, \mathit{red}, \mathit{black})\\
&\quad + [\mathit{color} = \textsc{Black}]\ \overline{\mathit{black}}.\mathit{Observer}(\mathit{ball}, \mathit{red}, \mathit{black})\ )
\end{aligned}$$

$$\begin{aligned}
\mathit{BallGenerator}(\mathit{ball}) = (\ \ &\tau.\overline{\mathit{ball}}\ \textsc{Red}.\mathit{BallGenerator}(\mathit{ball})\\
+\ &\tau.\overline{\mathit{ball}}\ \textsc{Black}.\mathit{BallGenerator}(\mathit{ball})\ )
\end{aligned}$$

but we can easily obtain an equivalent encoding without matching,

$$\begin{aligned}
&\mathit{Observer}(\mathit{ball}, \mathit{red}, \mathit{black}) = \mathit{ball}(\mathit{color}).\mathit{color}(\mathit{redcolor}, \mathit{blackcolor}).\\
&\quad (\ \ \mathit{redcolor}.\overline{\mathit{red}}.\mathit{Observer}(\mathit{ball}, \mathit{red}, \mathit{black})\\
&\quad + \mathit{blackcolor}.\overline{\mathit{black}}.\mathit{Observer}(\mathit{ball}, \mathit{red}, \mathit{black})\ )
\end{aligned}$$

$$\begin{aligned}
&\mathit{BallGenerator}(\mathit{ball}) = \overline{\mathit{ball}}(\mathit{new}).\overline{\mathit{new}}(\mathit{red}, \mathit{black}).\\
&\quad (\ \tau.\mathit{red}.\mathit{BallGenerator}(\mathit{ball}) + \tau.\mathit{black}.\mathit{BallGenerator}(\mathit{ball})\ )
\end{aligned}$$

and now if we specify the output port of the Observer using Definition 3.3, we have

$$\mathit{Output}(\mathit{red}, \mathit{black}) = \tau.\overline{\mathit{red}}.\mathit{Output}(\mathit{red}, \mathit{black}) + \tau.\overline{\mathit{black}}.\mathit{Output}(\mathit{red}, \mathit{black})$$

where the matching operations in the original Observer have been abstracted by silent transitions.

One may ask whether it is possible to obtain these role specifications automatically. Notice that Definition 3.3 defines roles up to weak bisimilarity. Taking into account the additional conditions for role correctness in Definition 3.5, we would only reject those roles which do not succeed or fail when the corresponding component does (as shown in the example previous to the definition of correctness), but given a component and a partition of its free names, a set of suites or categories of weak bisimilar roles can be obtained, all of them satisfying the conditions for correctness. Finding out which is the best role in each suite for representing meaningfully this partial view of the component is equivalent to find out the best understandable canonical form within a category of weak bisimilar agents. Though this process may be carried out with tool support, probably it cannot be fully automated, requiring certain supervision or guidance.

Anyway, some useful hints can be provided. The application of these hints will lead to the obtention of meaningful roles from a component. As we have seen, some component actions are transformed into τ-actions by means of hiding. We can obtain a weak bisimilar version of the role simply by removing most of these τ-actions. This can be done in role *Input* of Example 3.4. Its intermediate silent transitions could be omitted resulting in:

$$\mathit{Input}(i_1, i_2) = i_1(x).\mathit{Input}(i_1, i_2) + i_2(y).\mathit{Input}(i_1, i_2)$$

However, some of these internal actions may stand for local choices and they cannot be removed without affecting the behavior described in the role. This is the case of the role *Output* of the *Translator*. If the leading τ-actions are omitted, the resulting agent is not weak bisimilar with that in Example 3.4.

Also recursion can be omitted when it is unguarded, or prefixed only by τ-actions. Then, we will obtain a simplified version of the original role. This kind of transformations has been applied in roles *Storage* and *Retrieval* of the *Buffer* in Example 3.6. The original agents could be:

$$\mathit{Storage}(put) = put(it).(\tau.\mathbf{0} \mid \mathit{Storage}(put) + \tau.\tau.\tau.\tau.\mathit{Storage}(put)$$

$$\mathit{Retrieval}(get) = \tau.(\tau.\mathbf{0} \mid \mathit{Retrieval}(get) + \tau.get.\tau.\overline{get}(it).\mathit{Retrieval}(get)$$

but we can arrive to the respectively weak bisimilar roles specified in Example 3.6 by removing all the silent transitions and unguarded recursions.

4 Process Compatibility in the π-calculus

The notion of compatibility that will be introduced in this section tries to formalize a way to recognize when two roles, specified by π-calculus agents, conform each other.

A formal characterization of compatibility is given in Definition 4.3 below. Roughly, we may say that two roles are compatible if they can engage in at least one transition (1), any local choice in one of them is supported by the other (2), and any pair of *complementary* transitions will lead to compatible agents (3,4). Intuitively, these are the conditions we need to ensure that no mismatch will occur when these roles are composed in parallel, representing the attachment of the corresponding components.

Compatible agents must be able to synchronize at least in one common complementary transition. This is a necessary but not sufficient condition for compatibility, preventing to consider compatible agents which cannot interoperate like $a(x).\mathbf{0}$ and $\bar{b}y.\mathbf{0}$, which would deadlock when attached to each other.

Definition 4.1 *An agent P provides an input for an agent Q if P', Q' exist such that*

(1) $P \overset{\bar{x}y}{\Longrightarrow} P'$ and $Q \overset{x(z)}{\Longrightarrow} Q'$, or
(2) $P \overset{\bar{x}(z)}{\Longrightarrow} P'$ and $Q \overset{x(z)}{\Longrightarrow} Q'$

Definition 4.2 (Synchronizable agents) *Two agents P and Q are synchronizable if P provides an input for Q or Q provides an input for P.*

Definition 4.3 (Relation of (ground) compatibility) *A binary relation $\mathcal{C}$ on agents is a* semi-compatibility *if $P \mathrel{\mathcal{C}} Q$ implies*

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

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.*

Remark 4.4 *Notice that if $P \sim \mathbf{0}$ but $P \not\equiv \mathbf{0}$ (e.g. $P = (a)a(x).\mathbf{0}$) we have that $\forall\, Q\ P \not\dot{\diamond}\, Q$.*

The different treatment of silent and non-silent transitions in Definition 4.3 deals with global and local choices. Consider the agents $R_1 = a(x).\mathbf{0} + b(y).\mathbf{0}$ and $R_2 = \tau.a(x).\mathbf{0} + \tau.b(y).\mathbf{0}$. Though they present the same input actions, these actions appear as global choices in R_1, while in R_2 the commitment to a particular action $a(x)$ or $b(y)$ is locally decided. This causes that some agents which are compatible with R_1 are not compatible with R_2. Consider, for instance, $\overline{R_2} = \tau.\overline{a}u.\mathbf{0} + \tau.\overline{b}v.\mathbf{0}$. We have that $R_1 \mathrel{\dot{\diamond}} \overline{R_2}$, since the former is able to accept any choice indicated in the latter, fulfilling condition (2) in Definition 4.3. On the other hand, $R_2 \not\dot{\diamond}\, \overline{R_2}$, since condition (2) is not fulfilled, as R_2 may proceed to $b(y).\mathbf{0}$, and $\overline{R_2}$ to $\overline{a}u.\mathbf{0}$, both performing silent transitions, and $b(y).\mathbf{0} \not\dot{\diamond}\, \overline{a}u.\mathbf{0}$, since they are neither successful nor synchronizable.

Global choices which are not complementary in both agents are ignored in Definition 4.3 above. Consider an agent that presents an undesirable behavior when engaged in a certain action c, as $\overline{R_3} = \overline{a}u.\mathbf{0} + \overline{b}v.\mathbf{0} + \overline{c}\textsc{Error}$. Then we have that $R_1 \mathrel{\dot{\diamond}} \overline{R_3}$ and $R_2 \mathrel{\dot{\diamond}} \overline{R_3}$, since action c will never take place when composing $\overline{R_3}$ with R_1 or R_2. However, an agent which could chose locally to engage in this action, like $\overline{R_4} = \overline{a}u.\mathbf{0} + \overline{b}v.\mathbf{0} + \tau.\overline{c}\textsc{Error}$ is not compatible with any of the agents above.

Definition 4.5 (Compatible agents) *P and Q are compatible, written $P \diamond Q$, if $P\sigma \;\dot{\diamond}\; Q\sigma$ for all substitutions σ that respect the distinction $fn(P) \cup fn(Q)$.*

In order to understand why we avoid substitutions binding 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 $P \mathrel{\dot{\diamond}} Q$ 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 $\bar{x}.\mathbf{0} \not\dot{\diamond}\, \mathbf{0}$.

The restriction set upon substitutions reflects the situation with real software components. If two components follow a certain protocol for their interaction, we must distinguish between the different channels or messages used in this protocol, if not communication would be impossible. The same applies to roles. The free names in a role stand for the different *words* used in the interaction with other roles, and these names must be distinguished. Notice that, when several roles are attached, their free names are restricted (see Definition 3.7), thus ensuring their distinction.

Like Milner's relation of bisimilarity, our definition of compatibility involves universal quantification over substitutions. However, following a strategy similar to that in [11] it is possible to define an efficient and automatizable transition system for the relation, which allows the development of analysis tools.

4.1 Properties of Compatibility

Our relation of compatibility is symmetric, but as it requires the presence of complementary actions, it does not satisfy several common properties like reflexivity or transitivity. Even if we abstract the sign of the actions, these properties are not satisfied, and there is no logical implication between compatibility and bisimilarity. Consider again the examples above, where $R_1 \diamond \overline{R_3}$ though they present different transitions, but $R_2 \not\diamond \overline{R_2}$. In fact, similarity is not well suited for our purposes, since we don't need the processes related to behave identically or to match exactly. Thus, compatibility is not defined with the idea of comparing processes, but to have a safe and flexible way to connect them, and this lack of common properties seems reasonable. That is, whereas bisimilarity is defined to see when two process simulate each other, compatibility looks for the "safe" composition of processes.

However, other desirable properties are satisfied by the relation of compatibility. First of all, equivalence should preserve compatibility. Let's consider the following agents: $P = \mathbf{0}$, $P'(a) = (a)a(x).\mathbf{0}$ and $Q = \tau.\mathbf{0}$. Though $P \sim P'$ and $P \diamond Q$ we have that $P' \not\diamond Q$. The reason is that, unfortunately, neither strong nor weak bisimulation can differentiate between successful and unsuccessful termination. Since, this distinction is crucial for our purposes, we define a slightly finer relation, $\mathbf{0}$-*simulation*, as follows.

Definition 4.6 *$\mathcal{S}_0$ is a $\mathbf{0}$-simulation, if $P\ \mathcal{S}_0\ Q$ satisfies the conditions in Definition 2.2 and also*

- *If $P \equiv \mathbf{0}$ then $Q \equiv \mathbf{0}$*

Obviously, $P \mathcal{S}_0 Q \implies P \mathcal{S} Q$. Then, definitions of strong and weak $\mathbf{0}$-bisimilarity on agents, respectively $\sim_0$ and $\approx_0$, and weak and strong $\mathbf{0}$-bisimilar processes can be derived. Now, we can derive from Definition 4.3 that both strong and weak **0**-bisimilarity preserve compatibility.

Theorem 4.7 *Let P and Q be two compatible processes. If $P' \approx_0 P$ and $Q' \approx_0 Q$ then,*

$$P' \diamond Q'$$

Proof. It can be directly derived from Definition 4.3. We only have to prove that $\diamond_{weak} = \{(P', Q') : \exists\, P, Q \,.\, P \approx_0 P' \wedge Q \approx_0 Q' \wedge P \diamond Q\}$ is a relation of compatibility. Since compatibility is symmetric, it is enough to prove that f $P \diamond Q$ and $R \approx_0 P$ then $R \diamond Q$. In other words, we must prove that $\diamond_{weak} = \{(R, Q) : \exists\, P \,.\, P \approx_0 R \wedge P \diamond Q\}$ is a relation of compatibility. Assume $R \diamond_{weak} Q$, then we check for the conditions in Definition 4.3.

(1) a) If R is not successful then, since $R \approx_0 P$, P is also not successful. Then, from $P \diamond Q$ and Def. 4.3.1 we have that P and Q are synchronizable. Thus, $\exists\, \alpha, P', Q'$. $P \overset{\alpha}{\Longrightarrow} P'$ and $Q \overset{\overline{\alpha}}{\Longrightarrow} Q'$ (where $\overline{\alpha}$ stands for an action complementary to α). Since $R \approx_0 P$ we have that $\exists\, R'$. $R \overset{\alpha}{\Longrightarrow} R'$. Hence, R and Q are synchronizable.

b) On the other hand, if Q is not successful, then from $P \diamond Q$ and Def 4.3.1 we have that $\exists\, \alpha, P', Q'$. $P \overset{\alpha}{\Longrightarrow} P'$ and $Q \overset{\overline{\alpha}}{\Longrightarrow} Q'$. Since $R \approx_0 P$ we have that $\exists\, R'$. $R \overset{\alpha}{\Longrightarrow} R'$. Hence, R and Q are synchronizable.

(2) a) If $R \xrightarrow{\tau} R'$, since $R \approx_0 P$ then $\exists\, P' \,.\, P \Longrightarrow P'$ and $R' \approx_0 P'$. On the one hand, if P' is P (no τ-transition is performed), then from $R' \approx_0 P'$ and $P' \diamond Q$, we infer $R' \diamond_{weak} Q$. On the other hand, if $P(\longrightarrow)^{+}P'$ (at least one τ-transition is performed), since $P \diamond Q$, we have (Def. 4.3.2) that $P' \diamond Q$. Again, from $R' \approx P'$ and $P' \diamond Q$, we infer $R' \diamond_{weak} Q$.

b) If $Q \xrightarrow{\tau} Q'$, from $P \diamond Q$ and Def. 4.3.2 we have that $P \diamond Q'$. From that and also $R \approx_0 P$ we infer that $R \diamond_{weak} Q'$.

(3) a) If $R \xrightarrow{x(w)} R'$ and $Q \xrightarrow{\bar{x}y} Q'$, since $R \approx_0 P$ then $\exists\, P' \,.\, P \Longrightarrow P'' \xrightarrow{x(w)} P''' \Longrightarrow P'$ and $R' \approx_0 P'$. Then, from $P \diamond Q$ and Def. 4.3.2 and 4.3.3 we have that $P'' \diamond Q$, $P'''\{y/w\} \diamond Q'$, and $P'\{y/w\} \diamond Q'$. From that and $R' \approx_0 P'$ we infer $R'\{y/w\} \diamond_{weak} Q'$

b) If $Q \xrightarrow{x(w)} Q'$ and $R \xrightarrow{\bar{x}y} R'$, since $R \approx_0 P$ then $\exists\, P' \,.\, P \overset{\bar{x}y}{\Longrightarrow} P'$ and $R' \approx_0 P'$. Then, from Def. 4.3.2 and 4.3.3 we infer that $P' \diamond Q'\{y/w\}$. From that and $R' \approx_0 P'$ we infer $R' \diamond_{weak} Q'\{y/w\}$.

(4) a) If $R \xrightarrow{x(w)} R'$ and $Q \xrightarrow{\bar{x}(w)} Q'$, since $R \approx_0 P$ then $\exists\, P' \,.\, P \Longrightarrow P'' \xrightarrow{x(w)} P''' \Longrightarrow P'$ and $R' \approx_0 P'$. Then, from $P \diamond Q$ and Def. 4.3.2 and 4.3.4 we have that $P'' \diamond Q$, $P''' \diamond Q'$, and $P' \diamond Q'$. From that and $R' \approx_0 P'$ we infer $R' \diamond_{weak} Q'$

b) If $Q \xrightarrow{x(w)} Q'$ and $R \xrightarrow{\bar{x}(w)} R'$, since $R \approx_0 P$ then $\exists\, P' \,.\, P \overset{\bar{x}(w)}{\Longrightarrow} P'$ and $R' \approx_0 P'$. Then, from Def. 4.3.2 and 4.3.4 we infer that $P' \diamond Q'$. From that and $R' \approx_0 P'$ we infer $R' \diamond_{weak} Q'$. ■

Compatibility is not a congruence with respect to all constructions in π-calculus, but some related properties hold. These properties can be used to simplify the analysis of the compatibility among processes.

Theorem 4.8

(a) *From* $P \diamond Q$ *infer* $\tau.P \diamond \tau.Q$,
(b) *From* $P\{y/w\} \diamond Q$ *infer* $x(w).P \diamond \overline{x}y.Q$,
(c) *From* $P \diamond Q$ *infer* $x(w).P \diamond \overline{x}(w).Q$,
(d) *From* $P_1 \diamond Q$ *and* $P_2 \diamond Q$ *infer* $P_1 + P_2 \diamond Q$,
(e) *From* $P_1 \diamond Q_1$ *and* $P_2 \diamond Q_2$ *infer* $P_1 \mid P_2 \diamond Q_1 \mid Q_2$,
when $fn(P_1) \cap fn(P_2) = \varnothing$ *and* $fn(Q_1) \cap fn(Q_2) = \varnothing$,

Proof. The proof is straightforward, so it is omitted here. As an example, property (d) is proven in Theorem 5.14. ■

4.2 Compatibility and successful composition

Compatibility must ensure that no mismatch will arise from the interaction of the agents involved. This is the aim of Proposition 4.9 below.

Proposition 4.9 *Let P and Q be compatible agents. Then we have that their attachment is successful.*

Proof. It can be directly derived from Definitions 3.7 and 4.3. We have to prove that if $P \diamond Q$ then $(fn(P) \cup fn(Q))(P \mid Q)$ is successful. Since all free names are restricted we can reformulate the attachment as $(\mathcal{N})(P \mid Q)$ for short, where $\mathcal{N}$ contains any name. The proof is done supposing that the attachment is not successful, and then finding a contradiction.

Suppose that $P \diamond Q$ but that $(\mathcal{N})(P \mid Q)$ is not successful. Hence, exists a process *Failure* such that $(\mathcal{N})(P \mid Q) \Longrightarrow \mathit{Failure}$, where $\mathit{Failure} \not\equiv \mathbf{0}$ and $\mathit{Failure} \stackrel{\tau}{\not\longrightarrow}$. The proof is done by induction on the number n of τ-transitions leading to *Failure*.

(1) *Base Case.* Suppose first $n = 0$. Then, from $(\mathcal{N})(P \mid Q) \stackrel{\tau}{\not\longrightarrow}$ we have both $P \stackrel{\tau}{\not\longrightarrow}$ and $Q \stackrel{\tau}{\not\longrightarrow}$. Now, from $(\mathcal{N})(P \mid Q) \not\equiv \mathbf{0}$ we have either $P \not\equiv \mathbf{0}$ or $Q \not\equiv \mathbf{0}$. Hence, P or Q are not successful. Since $P \diamond Q$, from Def.4.3.1 we have that $\exists\, \alpha$. $P \stackrel{\alpha}{\Longrightarrow}$ and $Q \stackrel{\overline{\alpha}}{\Longrightarrow}$ (where $\overline{\alpha}$ stands for an action complementary to α). Hence, $(\mathcal{N})(P \mid Q) \stackrel{\tau}{\longrightarrow}$, which is a contradiction.

(2) *Inductive hypothesis.* $\forall P', Q' \ . \ P' \diamond Q'$, if $(\mathcal{N})(P' \mid Q')(\xrightarrow{\tau})^k F$ with $k < n$, then either $F \ \xrightarrow{\tau}$ or $F \equiv \mathbf{0}$.

(3) *General Case.* Suppose that $(\mathcal{N})(P \mid Q) \ \xrightarrow{\tau} \ (\mathcal{N})(P' \mid Q') \ (\xrightarrow{\tau})^{n-1}$*Failure*. Then, the initial τ-transition is one of the following:

- $P \xrightarrow{\tau} P'$. Then, since $P \diamond Q$ we have that $P' \diamond Q$.
- $Q \xrightarrow{\tau} Q'$. Then, since $P \diamond Q$ we have that $P \diamond Q'$.
- $P \xrightarrow{x(w)} P'$ and $Q \xrightarrow{\bar{x}y} Q'$. Then, since $P \diamond Q$ we have that $P'\{y/w\} \diamond Q'$.
- $P \xrightarrow{x(w)} P'$ and $Q \xrightarrow{\bar{x}(w)} Q'$. Then, since $P \diamond Q$ we have that $P' \diamond Q'$.
- $Q \xrightarrow{x(w)} Q'$ and $P \xrightarrow{\bar{x}y} P'$. Then, since $P \diamond Q$ we have that $P' \diamond Q'\{y/w\}$.
- $Q \xrightarrow{x(w)} Q'$ and or $P \xrightarrow{\bar{x}(w)} P'$. Then, since $P \diamond Q$ we have that $P' \diamond Q'$.

Hence, if $(\mathcal{N})(P \mid Q) \xrightarrow{\tau} (\mathcal{N})(P' \mid Q')$ we have that $P' \diamond Q'$, and using the inductive hypothesis, we infer that either *Failure* $\xrightarrow{\tau}$ or *Failure* $\equiv \mathbf{0}$. ■

The following result goes one step beyond, showing the effect of combining a component with roles compatible with its own roles.

Theorem 4.10 *Let Comp be a component correctly specified by a set of roles $\{P_i\}_i$, which represent Comp in its attachment to several other components $\{Comp_i\}_i$. Let Q_i be the role that represents respectively each $Comp_i$ in its attachment to Comp. Assume that $\forall\, i \ P_i \diamond Q_i$. Then we have that*

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

is successful.

Proof. It can be derived from Definitions 3.3, and 3.5. Definition 3.5 ensures that the free names in $\{P_i\}_i$ are disjoint, but some Q_i may have additional free names that collide with other names in *Comp* or even in some other role in $\{Q_i\}_i$. However, these additional free names are restricted, thus ensuring the independence of the attachments of *Comp*. Then, from $\forall\, i \ P_i \diamond Q_i$ and Definitions 3.3 and 3.5, we can derive the success of the composition of *Comp* with the roles $\{Q_i\}_i$. ■

Notice that the result above refers to the composition of a component with compatible roles, but not to its composition with the corresponding components. In fact, if two components are attached we cannot derive that their parallel composition is successful, since these components may in turn be connected to other components in the system. Furthermore, local analysis of compatibility in system attachments cannot prove either that the whole system is successful, since deadlock could arise from the global interaction of a set of components whose roles are compatible. However, the compatibility of system attachments serves to prove that no crash will arise from a behavioral mismatch between the interfaces of the interconnected components. This is enough to prove the composability of a certain system from its components and also the reusability of a certain component in a system different from that it was originally developed for.

Example 4.11 *Consider again the Buffer in Example 3.6, and two additional components: a Producer and a Consumer, whose behavior is represented by the roles:*

$$Producer(put) = \overline{put}(item).Producer(put)$$

$$Consumer(get) = \overline{get}.get(item).Consumer(get)$$

From Definition 4.3, it is trivial to find out that $Storage(put) \diamond Producer(put)$ *and* $Retrieval(get) \diamond Consumer(get)$. *Hence, from Theorem 4.10 we have also that*

$$Producer(put) \;\mid\; Buffer(put, get) \;\mid\; Consumer(get)$$

is successful. Therefore those components can be safely composed to build a Producer/Consumer system

Now, we can define a *composable architecture*, formed by the composition of several components, as a set of attachments between compatible pairs of roles of these components. This definition allows the extension of the results of theorems 4.9 and 4.10 to architectures.

Definition 4.12 (Composable Architecture) *Consider a binary architecture* Ψ *under the conditions of Definition 3.8.* Ψ *is a composable architecture if* $\forall\, P, Q \in \mathcal{R}oles$, *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 defined architectures as sets of attachments between pairs of roles, committing ourselves to binary relations. However, a more general definition could be considered, in which attachments involved more that two roles. This will be the aim of the next section.

4.3 Compatibility for groups of roles

The relation of compatibility in Definition 4.3 refers only to pairs of roles. However, compatibility can be extended to groups of more than two roles, allowing the analysis of more complex attachments, as they were defined in Definition 3.7.

Definition 4.13 (Set of synchronizable agents) *A set of agents $\mathcal{P}$ is synchronizable if $\exists\, P, Q \in \mathcal{P}$ such that P provides an input for Q.*

Definition 4.14 (Ground compatibility for a set of agents) *A set of agents $\mathcal{P}$ is ground compatible if*

(1) if $\exists\, P \in \mathcal{P}.\ P$ is not successful then $\mathcal{P}$ is a set of synchronizable agents,
(2) if $\exists\, P \in \mathcal{P}.\ P \xrightarrow{\tau} P'$ then $(\mathcal{P} - \{P\}) \cup \{P'\}$ is ground compatible,
(3) if $\exists\, P, Q \in \mathcal{P}.\ P \xrightarrow{x(w)} P'$ and $Q \xrightarrow{\bar{x}y} Q'$ then $(\mathcal{P} - \{P, Q\}) \cup \{P'\{y/w\}, Q'\}$ is ground compatible.
(4) if $\exists\, P, Q \in \mathcal{P}.\ P \xrightarrow{x(w)} P'$ and $Q \xrightarrow{\bar{x}(w)} Q'$ then $(\mathcal{P} - \{P, Q\}) \cup \{P', Q'\}$ is ground compatible.

Definition 4.15 (Compatibility for a set of agents) *A set of agents $\mathcal{P} = \{P_i\}_i$ is compatible if $\mathcal{P}\sigma = \{P_i\sigma\}_i$ is ground compatible for all substitutions σ that respect the distinction $\cup_i fn(P_i)$.*

Proposition 4.16 *Let $\mathcal{P}$ be a set of compatible agents. Then, their attachment is successful.*

Proof. It can be directly derived from Definitions 3.7 and 4.14. ■

Example 4.17 *Consider a certain component which uses our Buffer as a temporary store for some data that it produces and that will be required afterwards. The interface of this component could be represented by the role ProdCons (from Producer/Consumer):*

$$\begin{aligned} ProdCons(put, get) = {} & \overline{put}(item).ProdCons(put, get) \\ & + \overline{get}.get(item).ProdCons(put, get) \end{aligned}$$

Now we have an attachment between three different roles: ProdCons, Storage, and Retrieval. Using Definition 4.14 we can find out that these agents are compatible. Thus, their attachment is successful.

5 Inheritance and Extension of Behavior

The relation of compatibility among roles establishes the conditions for safe composition. However, in order to promote component and architecture reuse, it would be very interesting if we could check whether a certain existing component can be used in any context or architecture where another one appears. This idea is related to the concept of inheritance in the object-oriented paradigm, which refers to a relation among object classes by which a *child* class inherits the properties declared by its *parents*, while adding its own properties. Inherited properties may be redefined, usually under certain restrictions. Inheritance is a natural precondition for polymorphism, allowing dynamic replacement of an object by a derived version in any context where the original object appeared, and it promotes both reusability and incremental development.

Hence, we have defined a relation of agent inheritance in the context of the π-calculus, with the requirement that inheritance preserves compatibility, allowing safe replacement in any architecture of a component by a derived version whose roles inherit from those of the former. In our approach, roles represent the behavior of the corresponding components, i.e. how they react to external stimuli. Thus, a child role agent must maintain its parents' behavior, while redefinition and addition of behavior must be restricted in order to ensure that the parent component may be replaced by a derived version while maintaining compatibility. We consider inheritance as a form of strengthening the reliability of a role. Thus, derived roles are more predictable than their parents, by making fewer local choices, (and we call this *role inheritance*), while they may also offer new globally chosen behavior (which we call in turn *role extension*).

A compatibility-preserving relation of inheritance among processes requires the fulfillment of several conditions, related to inheritance of parent's behavior, redefinition, and extension. For this reason, the relation will be introduced in several steps.

Definition 5.1 (Semantics-preserving inheritance) *A binary relation $\mathcal{H}$ on agents is a relation of semantics-preserving inheritance if $R\mathcal{H}P$ implies that*

(1) if $P \xrightarrow{\bar{x}y} P'$ *then* $\exists\, R'.\ R \xrightarrow{\bar{x}y} R'$ *and* $R'\mathcal{H}P'$

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

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

Semantics-preserving requires that any globally chosen behavior offered by the parent agent is also present in the child. However, no condition is imposed over τ-actions, thus local choices may be converted into global by the child agent, or even suppressed. Assume that $P\diamond = \{Q\ :\ P \diamond Q\}$ is the set of processes which are compatible with a certain P. Assume $R\mathcal{H}P$. Then, the conditions above implies that any process in $P\diamond$ which takes a decision over global choices in P will be also compatible with R, while some processes not in $P\diamond$ will be in $R\diamond$, since some of the requirements of P (i.e. some of its local decisions) may be relaxed or suppressed in R. Hence, $P\diamond \subseteq R\diamond$.

Example 5.2 *Consider the agent $P(a) = a(x).\mathbf{0}$. Since P offers action $a(x)$ as a global choice, some agent $Q \in P\Diamond$, may commit to the complementary action by a local choice, for instance $Q(a,b) = \tau.\bar{a}(u).\mathbf{0} + \bar{b}(v).\mathbf{0}$. Hence, any agent which wishes to inherit from P must preserve the action $a(x)$ as a global choice. For instance, $R(a,c) = a(x).\mathbf{0} + c(z).\mathbf{0}$. Attending to Definition 5.1, we have that $R\mathcal{H}P$ and also $R \dot{\Diamond} Q$. Furthermore, some agents, like $S(a,c) = \tau.\bar{a}(u).\mathbf{0} + \tau.\bar{c}(w).\mathbf{0}$ which were not compatible with P are now compatible with R.*

Semantics-preserving is a necessary but not sufficient condition for inheritance, as shows the following example.

Example 5.3 *Consider the agents $P(a,b) = a(x).\mathbf{0} + \tau.b(y).\mathbf{0}$, $Q(b,c) = \bar{b}(u).\mathbf{0} + \bar{c}(v).v.\mathbf{0}$, and $R(a,b,c) = a(x).\mathbf{0} + b(y).\mathbf{0} + c(z).\mathbf{0}$. We have that $P \dot{\Diamond} Q$, and also that $R\mathcal{H}P$, attending to Definition 5.1. However, $R \not{\dot{\Diamond}} Q$, since $R \xrightarrow{c(v)} \mathbf{0}$, $Q \xrightarrow{\bar{c}(v)} v.\mathbf{0}$, and $\mathbf{0} \not{\dot{\Diamond}} v.\mathbf{0}$.*

Hence, we must impose additional conditions to semantics-preserving inheritance, which can be extended as follows:

Definition 5.4 (Non-extensible inheritance) *A binary relation $\mathcal{H}$ on agents is a relation of non-extensible inheritance if $R\mathcal{H}P$ implies the conditions in Definition 5.1, 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 $R \xrightarrow{x(y)} R'$ and $y \notin n(P,R)$ then $\exists\, P'.\ P \Longrightarrow\xrightarrow{x(y)} P'$
and $\forall\, w, R'\{w/y\}\mathcal{H}P'\{w/y\}$
(4) if $R \xrightarrow{\bar{x}(y)} R'$ and $y \notin n(P,R)$ then $\exists\, P'.\ P \Longrightarrow\xrightarrow{\bar{x}(y)} P'$ and $R'\mathcal{H}P'$

The conditions above require that the child agent R does not extend its parent P by offering new local or even global choices. (However, notice that the child roles may have converted some of its parent local choices into global.) The reason for this restriction to extension of behavior is that, as shown in Example 5.3, any new transition in R may interact with a complementary transition in a certain $Q \in P\Diamond$, where this transition was not considered when analyzing the compatibility of P and Q, since Definition 4.3 refers only to common complementary transitions in both agents. Observe that now, for the agents in Example 5.3, we have that R does not inherit from P. Thus, in order to preserve compatibility, our definition of inheritance must be very restrictive. However this restrictions will be overcome in the Definition 5.12 of agent extension.

Once again, non-extensible inheritance is a necessary condition for inheritance, but two additional conditions are required.

Definition 5.5 (Relation of inheritance) *A binary relation $\mathcal{H}$ on agents is a relation of inheritance if $R\mathcal{H}P$ implies the conditions in Definitions 5.1 and 5.4, and also*

(1) if $P \equiv \mathbf{0}$ then $R \equiv \mathbf{0}$
(2) if $P \xrightarrow{\tau}$ then $\exists\, P', R'.\ P \xrightarrow{\tau} P'$ and $R \Longrightarrow R'$ and $R'\mathcal{H}P'$

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

The first condition states which agents inherit from the inaction. The second one is less intuitive. It refers to τ-actions in the parent agent (which were not considered in Definition 5.1), and indicates that at least one of them must be inherited by the child role. This is a sufficient condition for maintaining R synchronizable with any agent in $P\diamond$, since only transitions complementary to local decisions in P are required for any agent $Q \in P\diamond$.

Example 5.6 *Consider now the agents $P(a,b) = a(x).\mathbf{0} + \tau.b(y).\mathbf{0}$, and $R(a) = a(x).\mathbf{0}$, which fulfill all conditions for inheritance but that of Definition 5.5.2. We have that $R \dot{\diamond} Q$ for most $Q \in (\diamond P)$, for instance $Q(a,b) = \overline{a}(u).\mathbf{0} + \overline{b}(v).\mathbf{0}$, but for $Q'(b) = \overline{b}(v).\mathbf{0}$ we have that $P \dot{\diamond} Q'$ but $R \not\dot{\diamond} Q'$. On the contrary, for $P'(a,b) = \tau.a(x).\mathbf{0} + \tau.b(y).\mathbf{0}$ 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 5.7 (Inheritance of behavior) *R inherits from P, written $R \triangleright P$, if $R\sigma \dot{\triangleright} P\sigma$ for every substitution σ that respects the distinction $fn(P) \cup fn(R)$.*

Theorem 5.9 below shows that inheritance of behavior preserves compatibility. However, an additional condition is necessary to prove the theorem, and for this reason, we reject processes that are *semantically divergent*, which –as considered in [13]– are those processes that may present an infinite sequence of local computations (represented by τ-actions). Since we are interested in checking interactions among roles, this is not a restriction, as shown in the following example:

Example 5.8 *Consider the agent $Comp(a,b) = a(x).Comp(a,b) + b(y).\mathbf{0}$ and two roles $Role_1(a) = a(x).Role_1(a) + \tau.\mathbf{0}$, and $Role_2(b) = \tau.Role_2(b) + b(y).\mathbf{0}$, obtained according to Definitions 3.3 and 3.5. Though $Comp(a,b)$ is not divergent, $Role_2$ may perform an infinite trace of τ-actions, but we can always find a different representation of $Comp(a,b)/_a$, in this case $Role'_2(b) = b(y).\mathbf{0}$, without infinite traces of τ-actions. Notice that $Role'_2 \approx Role_2$, and also that $\{Role_1, Role'_2\}$ is a correct set of roles for $Comp$.*

Theorem 5.9 *Let P and Q be two processes, where P does not present any infinite trace of τ-actions. Let $P \diamond Q$. Let $R \triangleright P$. Then we have that $R \diamond Q$.*

Proof. It can be derived from Definitions 4.3 and 5.5. We only have to prove that $\Diamond_{her} = \{(R, Q) \ : \ \exists\, P \ . \ P \Diamond Q \wedge R \ \rhd \ P\ \}$ is a relation of compatibility. Assume $R \Diamond_{her} Q$, then we check for the conditions in Definition 4.3.

(1) a) If R is not successful, then from Def. 3.1 $\exists\, R' \ . \ R \Longrightarrow R'$, where $R' \not\equiv \mathbf{0}$ and $R' \overset{\tau}{\not\longrightarrow}$.

Suppose first that R' is R. Since $R \rhd P$, $R \not\equiv \mathbf{0}$, and $R' \overset{\tau}{\not\longrightarrow}$, from Def. 5.5 we infer that $P \not\equiv \mathbf{0}$ and $P \overset{\tau}{\not\longrightarrow}$, i.e. P is not successful. Then, from $P \Diamond Q$ and Def. 4.3.1 we have that P and Q are synchronizable. Therefore, $\exists\, \alpha \ . \ P \xrightarrow{\alpha} P'$ and $Q \overset{\overline{\alpha}}{\Longrightarrow} Q'$, (where $\overline{\alpha}$ stands for an action complementary to α). Now, from $P \xrightarrow{\alpha} P'$ and Def. 5.1 we have that $R \xrightarrow{\alpha} R'$. Hence, R and Q are synchronizable.

Suppose now that R' is not R, i.e. $R(\xrightarrow{\tau})^n R'$, with $n > 0$. From $R \rhd P$ and Def. 5.4.1 we have that $\exists\, P' \ . \ P(\xrightarrow{\tau})^n P'$ and $R' \rhd P'$. Again, $R' \not\equiv \mathbf{0}$, $R' \overset{\tau}{\not\longrightarrow}$ and Def. 5.5 imply that $P' \not\equiv \mathbf{0}$ and $P' \overset{\tau}{\not\longrightarrow}$. Since $P \Diamond Q$, from Def. 4.3.2 we have that $P' \Diamond Q$ and, as P is not successful, from Def. 4.3.1, $\exists\, \alpha \ . \ P' \xrightarrow{\alpha} P''$ and $Q \overset{\overline{\alpha}}{\Longrightarrow} Q'$, (without τ-transitions from P'). From that, $R' \rhd P'$, and Def. 5.1, we infer that $R' \xrightarrow{\alpha} R''$. Hence, $R \overset{\alpha}{\Longrightarrow} R''$, and R and Q are synchronizable.

b) If Q is not successful, then from Def. 3.1 $\exists\, Q' \ . \ Q \Longrightarrow Q'$, where $Q' \not\equiv \mathbf{0}$ and $Q' \overset{\tau}{\not\longrightarrow}$. Since $P \Diamond Q$, from Def. 4.3.2 we have that $P \Diamond Q'$, and as Q' is not successful, we have that P and Q' are synchronizable. Therefore $\exists\, \alpha \ . \ P \overset{\alpha}{\Longrightarrow} P''$ and $Q' \xrightarrow{\alpha} Q''$ (without τ-transitions from Q').

Suppose first that $P \overset{\alpha}{\Longrightarrow} P''$ is $P \xrightarrow{\alpha} P''$. Then, from Def. 5.1 we have that $R \xrightarrow{\alpha} R''$. Therefore, R and Q' are synchronizable, and also R and Q are synchronizable.

Suppose now that $P \xrightarrow{\tau}\overset{\alpha}{\Longrightarrow} P''$ (i.e. there is at least one τ-transition between P and P''). From Lemma 5.10 below we have that $\exists\, P', R'. \ P \Longrightarrow P', R \Longrightarrow R', R' \rhd P'$ and $P' \overset{\tau}{\not\longrightarrow}$. Then, from $P \Diamond Q'$ we have that $P' \Diamond Q'$. Since Q' is not successful, we have that $\exists\, \beta \ . \ P' \xrightarrow{\beta}, Q' \xrightarrow{\beta}$ (without τ-transitions). Then, as $R' \rhd P'$, from Def. 5.1 we have that $R' \xrightarrow{\beta}$. Hence we have both R' and Q' are synchronizable, and R and Q are synchronizable.

(2) a) If $R \xrightarrow{\tau} R'$ then, from $R \rhd P$ and Def. 5.4.1, we have that $\exists\, P' \ . \ P \xrightarrow{\tau} P'$ and $R' \rhd P'$. Since $P \Diamond Q$, from Def. 4.3.2, we have that $P' \Diamond Q$. Hence, $R' \Diamond_{her} Q$.

b) If $Q \xrightarrow{\tau} Q'$ then, from $P \Diamond Q$ and Def. 4.3.2, we have that $P \Diamond Q'$. From that and $R \rhd P$, then we infer $R \Diamond_{her} Q'$.

(3) a) If $R \xrightarrow{x(w)} R'$ and $Q \xrightarrow{\bar{x}y} Q'$, from $R \rhd P$ and Def. 5.4.3, we have that $\exists\, P' \ . \ P \Longrightarrow P'' \xrightarrow{x(w)} P'$ and $\forall\, z \ R'\{z/w\} \rhd P'\{z/w\}$, where possibly P'' is P. Then, from Def. 4.3.2 we have that $P'' \Diamond Q$. From Def. 4.3.3, and $Q \xrightarrow{\bar{x}y} Q'$ we have that $P'\{y/w\} \Diamond Q'$. In particular, for $z = y$ we have

that $R'\{y/w\} \rhd P'\{y/w\}$. Hence, $R'\{y/w\} \diamond_{her} Q'$.

b) If $Q \xrightarrow{x(w)} Q'$ and $R \xrightarrow{\bar{x}y} R'$, from $R \rhd P$ and Def. 5.4.2 we have that $P \Longrightarrow P'' \xrightarrow{\bar{x}y} P'$ where $R' \rhd P'$ and possibly P'' is P'. Then, from $P \diamond Q$ and Def. 4.3.2 and 4.3.3 we have that $P'' \diamond Q$ and also $P' \diamond Q'\{y/w\}$. Hence, $R' \diamond_{her} Q'\{y/w\}$.

(4) a) If $R \xrightarrow{x(w)} R'$ and $Q \xrightarrow{\bar{x}(w)} Q'$, from $R \rhd P$ and Def. 5.4.3, we have that $\exists\, P' \,.\, P \Longrightarrow P'' \xrightarrow{x(w)} P'$ and $\forall\, z\ R'\{z/w\} \rhd P'\{z/w\}$, where possibly P'' is P. Then, from Def. 4.3.2 we have that $P'' \diamond Q$. From Def. 4.3.4, and $Q \xrightarrow{\bar{x}(w)} Q'$ we have that $P' \diamond Q'$. In particular, for $z = w$ we have that $R' \rhd P'$. Hence, $R' \diamond_{her} Q'$.

b) If $Q \xrightarrow{x(w)} Q'$ and $R \xrightarrow{\bar{x}(w)} R'$, from $R \rhd P$ and Def. 5.4.4 we have that $P \Longrightarrow P'' \xrightarrow{\bar{x}(w)} P'$ where $R' \rhd P'$ and possibly P'' is P'. Then, from $P \diamond Q$ and Def. 4.3.2 and 4.3.4 we have that $P'' \diamond Q$ and also $P' \diamond Q'$. Hence, $R' \diamond_{her} Q'$. ■

Lemma 5.10 *Let $R \rhd P$, where P does not present any infinite trace of τ-actions, and $P \xrightarrow{\tau}$. Then, $\exists\, P', R' \,.\, P \Longrightarrow P', R \Longrightarrow R', R' \rhd P'$ and $P' \not\xrightarrow{\tau}$.*

Proof. It can be derived from Def. 5.5. Under the conditions of the Lemma, we have that $\exists\, P_1, R_1 \,.\, P \xrightarrow{\tau} P_1, R \Longrightarrow R_1$ and $R_1 \rhd P_1$. Then, while $P_i \xrightarrow{\tau}$ we apply the definition again, obtaining P_{i+1}, R_{i+1}, where $R_{i+1} \rhd P_{i+1}$. Since P does not present any infinite trace of τ-actions, $\exists\, P_n, R_n \,.\, R_n \rhd P_n$ and $P_n \not\xrightarrow{\tau}$. ■

Thus, inheritance preserves compatibility, and a single proof of inheritance ensures that any 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.

Example 5.11 *A typical example of SA is that of Client/Server systems. Such an architecture is composed of two components –Client and Server– which behave as indicate the roles:*

$$\begin{aligned}&\mathit{Client}(\mathit{request}) = \overline{\mathit{request}}(\mathit{reply}, \mathit{error}).\\ &\quad (\ \mathit{reply}(\mathit{service}).\mathit{Client}(\mathit{request}) + \mathit{error}.\mathit{Client}(\mathit{request})\ )\end{aligned}$$

$$\begin{aligned}&\mathit{Server}(\mathit{request}) = \mathit{request}(\mathit{reply}, \mathit{error}).\\ &\quad (\ \tau.\overline{\mathit{reply}}(\mathit{service}).\mathit{Server}(\mathit{request}) + \tau.\overline{\mathit{error}}.\mathit{Server}(\mathit{request})\ )\end{aligned}$$

The Client requests a service, and either obtains it or gets an error. (Notice that mobility allows us to use private reply and error links in each request).

On the other hand, the Server may fail to serve some of the requests (local choices are used to represent this internal decision). Using definition 4.3 it is trivial to find out that Client $\Diamond$ Server.

Suppose now a component which behaves as describes role Client' below, crashing when an error is received.

$$\begin{aligned}&\mathit{Client}'(\mathit{request}) = \overline{\mathit{request}}\ (\mathit{reply}, \mathit{error}).\\ &\quad (\ \mathit{reply}(\mathit{service}).\mathit{Client}'(\mathit{request}) + \mathit{error}.\mathbf{0}\ )\end{aligned}$$

This component is not compatible with our Server (which can be proved using Definition 4.3 again), so we have to develop a fault-tolerant server, which we call FTServer, wrapping our server with a component FrontEnd which collects requests from the Client and retransmits them to the server until the service is obtained.

$$\mathit{FTServer}(\mathit{request}) = (\mathit{server})(\ \mathit{FrontEnd}(\mathit{request}, \mathit{server})\ \mid\ \mathit{Server}(\mathit{server})\ )$$

$$\begin{aligned}&\mathit{FrontEnd}(\mathit{request}, \mathit{server}) =\\ &\quad \mathit{request}(\mathit{reply}, \mathit{error}).\mathit{FTService}(\mathit{request}, \mathit{server}, \mathit{reply})\end{aligned}$$

$$\begin{aligned}&\mathit{FTService}(\mathit{request}, \mathit{server}, \mathit{reply}) = \overline{\mathit{server}}(\mathit{rep}, \mathit{err}).\\ &\quad (\ \ \mathit{rep}(\mathit{service}).\overline{\mathit{reply}}\ \mathit{service}.\mathit{FrontEnd}(\mathit{request}, \mathit{server})\\ &\quad + \mathit{err}.\mathit{FTService}(\mathit{request}, \mathit{server}, \mathit{reply})\ )\end{aligned}$$

From the point of view of the server, FrontEnd behaves as a client, so Client is a correct role of FrontEnd/$\{\mathit{request}\}$. This ensures the composability of the FTServer from its subcomponents; since Client $\Diamond$ Server, we can compose safely FrontEnd with our server component.

On the other hand, agent Server' below is a correct role of FTServer, and from Definition 4.3 we find that Client' $\Diamond$ Server'.

$$\mathit{Server}'(\mathit{request}) = \mathit{request}(\mathit{reply}, \mathit{error}).\overline{\mathit{reply}}(\mathit{service}).\mathit{Server}'(\mathit{request})$$

In addition, it can be proved using Definition 5.5 that Server' $\rhd$ Server, since it has suppressed the local choices of the latter. Hence, Server' $\Diamond$ Client, and we can claim that the corresponding component FTServer 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 complex systems.

The relation of inheritance presented above is too restrictive: Definition 5.1 states that the child role cannot extend its parents, adding new behavior or functionality. We can overcome these restrictions defining *extension* as follows:

Definition 5.12 (Extension of behavior) *R extends P, written $R \rhd\!\!\!\rhd P$, iff*

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

This definition relates extension to inheritance, and establishes the conditions which ensure that the extended role R can be successfully attached to any $Q \in P\diamond$. In order to preserve compatibility, additional behavior in the child agent R is restricted, ensuring that, when R and Q are attached, R will behave as P did (with the only difference that some local choices in P may be redefined or omitted in R). Additional functionality provided for the child agent R will not be used. However, this additional behavior of R may be successfully used in other contexts or architectures, even in combination with Q.

Example 5.13 *John, Paul, and Mary are friends. They usually meet in pairs to chat for a while. When they meet, they greet each other saying "hi", and agree on talking about a certain "topic". They part after saying "bye". John only talks about a single topic in each conversation. On the contrary, Mary can agree with her partner in changing to a new topic during the conversation. Finally, Paul seems to accept a change of topic, but he goes on talking about the same, hindering the conversation. Their behavior is specified by the following roles:*

$$\begin{aligned}
&John(hi, bye) = \overline{hi}(topic).JohnTalking(hi, topic, bye)\\
&JohnTalking(hi, topic, bye) = \tau.\overline{topic}.JohnTalking(hi, topic, bye)\\
&\qquad\qquad\qquad\qquad\qquad\quad + \tau.\overline{bye}.John(hi, bye)
\end{aligned}$$

$$\begin{aligned}
&Paul(hi, change, bye) = hi(topic).PaulTalking(hi, topic, change, bye)\\
&PaulTalking(hi, topic, change, bye) =\\
&\qquad\quad topic.PaulTalking(hi, topic, change, bye)\\
&\quad + \ change(newtopic).PaulTalking(hi, topic, change, bye)\\
&\quad + \ bye.Paul(hi, change, bye)
\end{aligned}$$

$$\begin{aligned}
&Mary(hi, change, bye) = \overline{hi}(topic).MaryTalking(hi, topic, change, bye)\\
&MaryTalking(hi, topic, change, bye) =\\
&\qquad\quad \tau.\overline{topic}.MaryTalking(hi, topic, change, bye)\\
&\quad + \ \overline{change}(newtopic).MaryTalking(hi, newtopic, change, bye)\\
&\quad + \ \tau.\overline{bye}.Mary(hi, change, bye)
\end{aligned}$$

Notice first that John $\diamond$ Paul, since the former will never change topics, and won't notice Paul's unkind behavior. However, Mary will notice it, and she is not compatible with Paul. Notice also that Mary $\not\rhd$ John, but Mary $\rhd\!\!\!\rhd$ John. Hence, we can conclude that (change)Mary $\diamond$ Paul, that is, provided Mary doesn't try to change topics, her behavior will be compatible with Paul's.

This example can be read with a different light if we look at John as a connection-oriented server (like a TCP server), in the sense that it uses a single channel (topic) for each session or conversation with a client. On the contrary, Mary acts like a connectionless, transaction-oriented server, since she may change the communication channel during the transmission (like an UDP server). Finally Paul is a somehow mischievous client.

Therefore, the relation of extension ensures safe replacement, but without changing the characteristics of the architecture where the replacement occurs. However, it is also possible that $R \diamond Q$ without restricting the additional behavior. In that case, the replacement of P by R implies a change in the characteristics of the architecture, and the additional behavior of R is used in the resulting system. Thus, the architecture in which the attachment of P and Q occurs describes a whole family of similar but not identical software products. We can obtain result similar to Theorem 5.9 for this family of systems with a common architectural pattern.

Theorem 5.14 *Let $P \diamond Q$. Let $R = P + P'$. If $P' \diamond Q$ we have that $R \diamond Q$.*

Proof. (Notice that this is also a proof for Theorem 4.8.d.) It can be derived from Definitions 4.3 and 5.5. We only have to prove that $\diamond_{ext} = \{(R, Q) : \exists\, P, P' \,.\, R = P + P' \wedge P \diamond Q \wedge P' \diamond Q\}$ is a relation of compatibility. Assume $R \diamond_{ext} Q$, then we check for the conditions in Definition 4.3.

(1) a)If R is not successful then either P or P' (or both) are not successful. Then, from $P \diamond Q$, $P' \diamond Q$ and Def. 4.3.1 we have that Q is synchronizable with both P and P'. Hence, we have that $R = P + P'$ is synchronizable with Q.

b) On the other hand, if Q is not successful, from $P \diamond Q$, $P' \diamond Q$ and Def. 4.3.1 we have that Q is synchronizable with both P and P'. Thus, $\exists\, \alpha, \beta \,.\, P \stackrel{\alpha}{\Longrightarrow}, P' \stackrel{\beta}{\Longrightarrow}, Q \stackrel{\overline{\alpha}}{\Longrightarrow}, Q \stackrel{\overline{\beta}}{\Longrightarrow}$ (where $\overline{\alpha}$, $\overline{\beta}$ stand for actions complementary to α and β, respectively). Hence, even in presence of τ-actions before α or β we have that $R = P + P'$ is synchronizable with Q.

(2) a) If $R \xrightarrow{\tau} R'$, then either $P \xrightarrow{\tau} R'$ or $P' \xrightarrow{\tau} R'$. In any case, from $P \diamond Q$, $P' \diamond Q$ and Def. 4.3.2 we have that $R' \diamond Q$. If we consider $R'' = R' + R'$ we will have that $R'' \diamond_{ext} Q$, while obviously $R'' \equiv R'$.

b) If $Q \xrightarrow{\tau} Q'$, from $P \diamond Q$, $P' \diamond Q$ and Def. 4.3.2 we have both $P \diamond Q'$ and $P' \diamond Q'$. Since $R = P + P'$, we infer $R \diamond_{ext} Q'$.

(3) a) If $R \xrightarrow{x(w)} R'$ and $Q \xrightarrow{\bar{x}y} Q'$, then $P \xrightarrow{x(w)} R'$ or $P' \xrightarrow{x(w)} R'$ (or both). In any case, from $P \diamond Q$, $P' \diamond Q$ and Def. 4.3.3 we have that $R'\{y/w\} \diamond Q'$. If we consider $R'' = R' + R'$ we will have that $R''\{y/w\} \diamond_{ext} Q'$, while obviously $R'' \equiv R'$.

b) Similarly, if $Q \xrightarrow{x(w)} Q'$ and $R \xrightarrow{\bar{x}y} R'$, then either P or P' (or both) will present a transition $\bar{x}y$ leading to R'. Then, from $P \diamond Q$, $P' \diamond Q$ and Def. 4.3.3 we have that $R' \diamond Q'\{y/w\}$. Again, if we consider $R'' = R' + R'$ we will have that $R'' \diamond_{ext} Q'\{y/w\}$, while obviously $R'' \equiv R'$.

(4) a) If $R \xrightarrow{x(w)} R'$ and $Q \xrightarrow{\bar{x}(w)} Q'$, then $P \xrightarrow{x(w)} R'$ or $P' \xrightarrow{x(w)} R'$ (or both). In any case, from $P \diamond Q$, $P' \diamond Q$ and Def. 4.3.4 we have that $R' \diamond Q'$. If we consider $R'' = R' + R'$ we will have that $R'' \diamond_{ext} Q'$, while obviously $R'' \equiv R'$.

b) Similarly, if $Q \xrightarrow{x(w)} Q'$ and $R \xrightarrow{\bar{x}(w)} R'$, then either P or P' (or both) will present a transition $\bar{x}(w)$ leading to R'. Then, from $P \diamond Q$, $P' \diamond Q$ and Def. 4.3.4 we have that $R' \diamond Q'$. Again, if we consider $R'' = R' + R'$ we will have that $R'' \diamond_{ext} Q'$, while obviously $R'' \equiv R'$. ■

Hence, once we have proved that an attachment among two roles P and Q is compatible, for any role $R = P + P'$ that extends P we only need to test the compatibility of the additional behavior P'. This result justifies the introduction of linguistic constructions related to inheritance at the level of a higher-order language, in order to hide the complexity of the relation of inheritance to software engineers.

6 Discussion

The importance of specification and analysis is more evident as software systems become more complex. However, there is a lack of methods and tools specifically developed for the specification and analysis of the structure of software systems. Traditionally, software architectures have been described in an informal manner, leading to problems in the development process.

In this work, we tried to show how the use of π-calculus, a well-known process algebra, can be a solution to some of these problems. Mobility is easily specified in π-calculus, which makes our proposal specially interesting for systems with evolving communication topology, such as open systems. We have given a definition of role in the context of π-calculus, and described how roles can be derived from the specification of components.

However, the main contributions of this work are the relations of compatibility and inheritance defined in the context of the π-calculus, together with the properties that we have proven are hold by these relations. 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 a relation of role inheritance and extension which preserves compatibility. This relation permits the replacement of components with specialized versions, maintaining the compatibility of the system with no need of checking the attachments modified by the replacement.

The examples included in this work are deliberately simple, and they are provided with the intention of illustrating the concepts described. More complex and therefore more interesting examples, including not binary architectures, can be found in [16] which contains a case study about a distributed auction showing the applicability of our work.

In the last years, SA has deserved active research interest [4], and a great number of proposals for architecture description, analysis and development have been presented. The basic idea which is behind all these proposals is that of focusing on identifying the main blocks or components for building software systems, and describing the interaction patterns that these components follow. However, some proposals [17,18] differentiate between components, described by a set of *ports*, and connectors, for gluing components, and described by a set of roles. From our point of view, the distinction between components and connectors is often subtle. Usually software artifacts share characteristics of components (they perform some computation) and connectors (they serve to interconnect other components). Furthermore, the composition of components and connectors would lead to hybrid composites with free ports and roles which could be classified neither as components nor as connectors. In order to maintain regularity and simplicity, we do not distinguish at the specification level between these categories, and both – components and connectors – are called generically *components*, and are described by a set of roles representing their interface. This choice also simplifies the formalization of the definitions of roles, compatibility, and inheritance, and the corresponding results about successful composition.

A complete discussion about all the proposals related to SA would require a separate work, but a good comparison on some of the most relevant can be found in [19,20]. Many of these proposals have a formal basis, which allows some kind of verification of the architectures described, either for checking their conformance with architectural styles, or for checking diverse properties of the architectures. Hence, process algebras, such as CSP [6], π-calculus [21], or ACP [22], and also other formalisms, such as the CHAM [23], TLA [24], Z [25], *posets* [26], and graph grammars [27], have been proposed as the formal basis for different architectural description languages and frameworks. We shall discuss now those which are closer to our approach.

The TOOLBUS architecture [22] uses interface specifications, which they call *scripts*, for specifying the behavior of components. These scripts are written in a timed extension of the process algebra ACP. However, their purpose is mainly descriptive, and not analytical, and they restrict themselves to a fixed architecture, while we try to address the description of arbitrary software architectures.

Rapide [26] is an event-based ADL in which the behavior of components is given by event patters that describe partially ordered set of events or *posets*. Then, simulation is used to check the consistency of interfaces. Each simulation results in a poset that represents one particular interaction among the components. Proper or correct orderings of events are described by imposing constraints on posets.

In [28], the π-calculus is used for defining the semantics of the ADL Darwin [21]. Although type checking was initially reduced in Darwin to name equivalence, later works [29] propose the description of the behavior of components using finite state Labeled Transition Systems, and their analysis by means of reachability analysis with the TRACTA framework [30]. Like in our approach, interface description using roles and hierarchical composition are used to minimize state explosion. However, the formalism used lacks expression of mobility and is not suitable for describing dynamic architectures, nor they consider aspects like inheritance or refinement of behavior. An interesting point of these works is that the authors claim that their approach allows to check not only for deadlock, but also for other safety and liveness properties, when these properties are conveniently specified as finite state machines. The later suggests an interesting research line which deserves further consideration.

Our definition of compatibility follows the ideas developed in [6], where CSP is proposed for the specification of the behavior of components in the ADL Wright. However, formalisms like CSP or CCS do not seem appropriate for the description of structures with changing communication topology. At most, CSP can be used in systems with a fixed number of configurations, at is shown in [31], but not in highly dynamic systems, where formalisms like π-calculus are best suited, making possible the specification of roles for dynamic systems. Therefore, our definition of compatibility must take into account mobility.

Allen and Garlan's work is based on the concept of refinement in CSP. In their paper, refinement, which could be roughly described as a relation of inclusion between processes, is used to define a relation of compatibility between asymmetric ports (representing components) and roles (representing connectors between components). A port is compatible with a role if the former refines the later. Our approach differs from theirs in that we do not distinguish between components and connectors; a connector is considered as a special kind of component, and like any other component is represented by a set of roles, partial abstractions of its interface. This leads to a much simpler and regular setting in which attachments are made between roles which present actions of *different sign*, representing components of different sign, too.

Another difference with [6] is that our approach gives a methodology for specifying the roles of a component, and need no transformation of the roles (e.g. like their deterministic versions of roles) for compatibility checking. Furthermore, their work does not address issues of inheritance or extension which in our work are used for defining the conditions of component substitutability preserving the compatibility of the architecture.

The notion of compatibility is present in some other relevant works, such as [32] or [33]. In [32] the CHAM is used for specifying software architectures and two different kinds of analysis are used for checking liveness properties of a certain architecture. However, they do not address issues of inheritance or extension. In [33], finite-state diagrams are used for the specification of what they call *protocols*, and relations of compatibility and protocol subtyping are also provided. While our approach leads to similar results, we overcome some limitations present in theirs. First, our proposal work addresses the specification and analysis of dynamic systems, while theirs can be applied only to static components. Second, we use name restriction to obtain modular specifications of components and roles, while in [33] 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 decisions. Finally, in their approach, if a protocol presents an output action, any compatible protocol must present a complementary input action, in the sense that what they call *undefined receptions* can't occur. However, we can check the compatibility of agents with different sets of free names and actions, allowing the combination of components which match only partially.

The notion of inheritance and extension of behavior presented in this paper has several analogies with the notion of Action Refinement in the context of process algebras [34]. The underlying motivation of both approaches is similar; both explore the relations among derived or refined versions of components and specifications. However, action refinement refers to a relation among agents by which an atomic action in one agent is replaced by a whole process in the other. Both agents describe the *same* component, but at different levels of abstraction, and action refinement can be used as a guide in the path from specification to implementation [35] by adding further details to components during the development process. On the other hand, our relation of inheritance supports the replacement of a component (usually within the same level of abstraction) with a *different* one whose roles describe weaker requirements or offer a larger functionality, in a similar way as inheritance is used in the object-oriented paradigm (which motivates the name of our relation). Hence, both approaches are complementary.

Our current work is related to the development of tools for checking the relations of compatibility and inheritance introduced in this work. In particular, we are working with prototypes derived from the Mobility Workbench [3]. Another promising research line consists on the analysis of other properties of software architectures, apart from deadlock-freedom. As shown in [30], if we specify in π-calculus both a component (or its roles, to be exact), and one certain property it must fulfill, it is possible to check if the behavior of the component satisfies the property by analyzing the compatibility of these agents.

Nevertheless, π-calculus is a low-level notation, which makes difficult its direct application to industrial-size software development. Hence, we are also developing LEDA [9], a higher-level Architecture Description Language which is based on the 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.

References

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

[2] C. Canal, L. Fuentes, J. M. Troya, A. Vallecillo, Extending CORBA interfaces with π-calculus for protocol compatibility, in: TOOLS Europe 2000, IEEE Computer Society, Mont Saint–Michel (France), 2000, pp. 208–225.

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

[4] D. Garlan, D. Perry, Special issue on software architecture, IEEE Transactions on Software Engineering 21 (4).

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

[6] R. Allen, D. Garlan, A formal basis for architectural connection, ACM Transactions on Software Engineering and Methodology 6 (3) (1997) 213–49.

[7] C. Canal, E. Pimentel, J. Troya, On the composition and extension of software systems, in: FSE'97 Workshop on Foundations of Component-Based Systems, Zurich (Switzerland), 1997, pp. 50–59.

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

[9] C. Canal, E. Pimentel, J. Troya, Specification and refinement of dynamic software architectures, in: Software Architecture, Kluwer Academic Publishers, 1999, pp. 107–126.

[10] R. Milner, The polyadic π-calculus. a tutorial, Tech. rep., University of Edinburgh (October 1991).

[11] D. Sangiorgi, A theory of bisimulation for the π-calculus, Tech. Rep. ECS-LFCS-93-270, University of Edinburgh (June 1993).

[12] M. Pistore, D. Sangiorgi, A partition refinement algorithm for the π-calculus, in: Computer Aided Verification CAV'96, no. 1102 in LNCS, Springer Verlag, 1996, pp. 38–49.

[13] L. Aceto, M. Hennessy, Termination, deadlock, and divergence, Journal of the ACM 39 (1) (1992) 147–187.

[14] G. Boudol, Notes on algebraic calculi of processes, in: Logics and Models of Concurrent Systems, no. 13 in NATO ASI series, K.Apt, 1987.

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

[16] C. Canal, E. Pimentel, J. Troya, Specification of interacting software components: a case study, in: II Jornadas Iberoamericanas de Ingenieria de Requisitos y Ambientes Software (IDEAS'99), San José (Costa Rica), 1999, pp. 381–392.

[17] M. Shaw, et al., Abstractions for software architecture and tools to support them, IEEE Transactions on Software Engineering 21 (4) (1995) 314–335.

[18] R. Allen, D. Garlan, Formalizing architectural connection, in: 16th International Conference on Software Engineering (ICSE'94), Sorrento (Italy), 1994, pp. 71–80.

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

[20] N. Medvidovic, R. N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions on Software Engineering 26 (1) (2000) 70–93.

[21] J. Magee, J. Kramer, Dynamic structure in software architectures, in: ACM Foundations of Software Engineering (FSE'96), San Francisco (USA), 1996, pp. 3–14.

[22] J. Bergstra, P. Klint, The discrete time toolbus — a software coordination architecture, Science of Computer Programming 31 (2–3) (1998) 179–203.

[23] P. Inverardi, A. L. Wolf, Formal specification and analysis of software architectures using the chemical abstract machine model, IEEE Transactions on Software Engineering 21 (4) (1995) 373–386.

[24] P. Ciancarini, M. Mazza, L. Pazzaglia, A logic for a coordination model with multiple spaces, Science of Computer Programming 31 (2-3) (1998) 205–229.

[25] G. Abowd, R. Allen, D. Garlan, Using style to understand descriptions of software architecture, in: ACM Foundations of Software Engineering (FSE'93), 1993.

[26] D. C. Luckham, et al., Specification and analysis of system architecture using rapide, IEEE Transactions on Software Engineering 21 (4) (1995) 336–355.

[27] D. LeMétayer, Describing software architecture styles using graph grammars, IEEE Transactions on Software Engineering 24 (7) (1998) 215–225.

[28] J. Magee, S. Eisenbach, J. Kramer, Modeling darwin in the π-calculus, in: Theory and Practice in Distributed Systems, no. 938 in LNCS, Springer Verlag, 1995, pp. 133–152.

[29] J. Magee, J. Kramer, D. Giannakopoulou, Behaviour analysis of software architectures, in: Software Architecture, Kluwer Academic Publishers, 1999, pp. 35–49.

[30] D. Giannakopoulou, J. Kramer, S. C. Cheung, Behaviour analysis of distributed systems using the tracta approach, Journal of Automated Software Engineering 6 (1) (1999) 7–35.

[31] R. Allen, R. Doucence, D. Garlan, Specifying and analyzing dynamic software architectures, in: ETAPS'98, Lisbon (Portugal), 1998.

[32] D. Compare, P. Inverardi, A. Wolf, Uncovering architectural mismatch in component behavior, Science of Computer Programming 33 (2) (1999) 101–131.

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

[34] L. Aceto, M. Hennessy, Adding action refinement to a finite process algebra, Information and Computation 115 (2) (1994) 179–247.

[35] R. Gorrieri, A. Rensik, Action refinement as an implementation relation, in: FASE'97, no. 1214 in LNCS, Springer Verlag, 1997, pp. 772–786.