A Calculus for Reconfigurable Component-Based Systems

Cidcley Teixeira de Souza

Federal Center of Technological Education of Ceará
Fortaleza, Ceará, Brazil, 60040-531
cidcley@cefetce.br

and

Paulo Roberto Freire Cunha

Federal University of Pernambuco
Recife, Pernambuco, Brazil, 50732-970
prfc@cin.ufpe.br

Abstract

The concept of reconfigurable systems is almost always restricted to the abstract design level, in which configuration languages are used to manipulate software connections. In this paper, we propose to deal with reconfiguration abstraction also in the formal specification level. We show that the process algebra π -calculus, widely applied to specify composable systems, is not suitable to capture all the properties required by these kinds of systems. To deal with these restrictions of π , we propose the $\mathcal{R}\pi$ -calculus. $\mathcal{R}\pi$ is a π -calculus extension that uses the concept of components as unit of specification. Its semantics allows for the reconfiguration of formal specifications in the same way configuration languages do.

Keywords: Components, Formal Methods, Reuse, Process Algebra.

1 Introduction

The development of large software systems is changing increasingly due to the use of reusable components developed by third parties [7, 10]. The use of such reusable components allows developers to reduce the system development time and costs. Moreover, there is a gain on flexibility, reliability and reusability of the final application, due to the (re)use of software components that had been already tested and validated [1], being the systems built by means of composition of these components.

Process algebras, like CCS [5] and π -calculus [6], are algebraic languages which support the compositional description of concurrent and distributed systems and the formal verification of their properties. The basic elements of any process algebra are their actions, which represent activities

carried out by the systems being modelled, and their operators, which are used to compose algebraic descriptions.

The π -calculus has been successfully applied for specifying configurations. Languages like Darwin [4, 2] use this process algebra as underlying semantics to capture the behavior of program configurations. Darwin separates the description of the system structure from the algorithms to describe individual processes. Thereby a process instantiation and reuse is permitted in different contexts. The structural description corresponds to the issue of *programming in the large* whereas the process description corresponds to the *programming in the small*.

However, we will show in this paper by means of Darwin examples, that π -calculus is not suitable for specifying reconfigurable systems, since we cannot build π -calculus specifications by having the properties required for this kind of systems. In that way, we propose an extension to this process algebra, termed $\mathcal{R}\pi$, that captures all properties required to build reconfigurable systems. The syntax of $\mathcal{R}\pi$ allows the definition of components as units of specification and, by means of its semantics, we can relax the synchronization discipline of π -calculus in such a way that it is possible to easily reuse and reconfigure these specifications.

The remainder of this paper is organized as follows. In section 2 we present an overview of π -calculus and the basic ideas of reconfigurable systems by means of the Darwin language. The use of π -calculus for specifying reconfigurable systems is presented in the section 3. Also in this section we present Darwin examples and a discussion about the suitability of π -calculus as a language for specifying reconfigurable systems. In section 4 we present the $\mathcal{R}\pi$ -calculus, an extension of π -calculus that allows the algebraic specification of reconfigurable systems. In section 6 we exemplify the applicability of $\mathcal{R}\pi$ by means of a classical reconfigurable application. Related works are presented in section 7. Finally, in section 8, we report some concluding remarks.

2 Basic Concepts

2.1 The π -calculus

The π -calculus [6] is an elementary calculus for describing and analyzing of concurrent systems with evolving communication structure. In this paper, we use the simple monadic form. A system in the π -calculus is a collection of independent processes which communicate via channels. Channels or links are referred to by name. Names are the most primitive entities in the calculus, they have no structure. There is an infinite number of names, represented here by lowercase letters. Processes are built from names as follows:

Actions :

$$\alpha ::= \begin{array}{l} a(x) \quad (Input) \\ \bar{a}y \quad (Output) \end{array}$$

1. $a(x)$ is called an *input prefix*. A name a is thought of as an input port of a process which contains it.
2. $\bar{a}y$ is called an *output prefix*. A name a is thought of as an output port of a process which contains it.

The agents ranged over by $P, Q, R, \dots$, are defined by

Agents :

$$\begin{array}{ll}
 P ::= & \emptyset \quad (\textit{Inaction}) \\
 & | \quad \alpha.P \quad (\textit{Prefix}) \\
 & | \quad Q + R \quad (\textit{Summation}) \\
 & | \quad Q \mid R \quad (\textit{Composition}) \\
 & | \quad (\nu x)Q \quad (\textit{Restriction})
 \end{array}$$

1. $\emptyset$ is the empty agent, which cannot perform any action.
2. $\alpha.P$ means that the process performs one of the α actions and then continues as P .
3. $Q + R$ represents an agent that can enact either Q or R .
4. $Q \mid R$ represents the combined behavior of Q and R executing in parallel.
5. $(\nu x)Q$ introduces a new name x with scope Q (binds all free occurrences of x in Q).

2.1.1 Operational Semantics

Computation in the π -calculus can be expressed by the following reduction rule:

$$(\dots + x(y).P_1 + \dots) \mid (\dots + \bar{x}z.P_2 + \dots) \rightarrow P_1\{z/y\} \mid P_2$$

Sending z along channel x reduces the left hand side to $P_1 \mid P_2$ with all free occurrences of y in P_1 replaced by z .

2.2 The Darwin Language

Darwin [4, 2] is a configuration language that allows distributed programs to be constructed from hierarchically structured specifications of the set of component instances and their interconnections. Composite component types are constructed from the primitive computational components and these in turn can be configured into more complex composite types. Components interact by accessing services.

2.2.1 Components and Services

Darwin views components in terms of both the services they provide to allow other components to interact with them and the services they require to interact with other components. For example, the component of Figure 1 is a filter component which provides a single service output and requires a single service input. The filled in circles represent services provided by a component and empty circles represent services required by a component. The type of the service is specified in angle brackets. In the example, the communication mechanism used to implement the service is a stream and the datatype communicated is **char**. Darwin does not interpret service type information, it is used by the underlying distributed platform.

Figure 1: Component in Darwin.

A component does not need to know the global names of external services or where they are to be found in the environment. Components may thus be implemented and tested independently of the rest of the system of which they will be a part. We call this property *context independency*. It allows the reuse of components during construction (through multiple instantiation) and simplifies replacement during maintenance.

3 Reconfigurable Systems in π -calculus

The purpose in modelling a system in the π -calculus is to provide a precise semantics for the language. In this section we use the Darwin language to show how to use π -calculus to this end. In this sense we show that a Darwin configuration program correctly specifies the set of primitive component instances and set of intercomponent bindings required at runtime. Darwin supports static checking to ensure only bindings between compatible requirements and provisions are allowed.

3.1 Connections

The connections in Darwin are modelled by the constructors concerned with requiring, providing and binding services. The declaration of a provided service, **provide** p , in Darwin is modelled in the π -calculus as the agent $Prov(p, s)$ which is accessed by the Darwin name p and manages the service s as follows:

$$Prov(p, s) \stackrel{\text{def}}{=} !p(x).\bar{x}s$$

The service s is simply the name or reference to a service which must be implemented by a component. Darwin is not concerned with how the service s is implemented, it is concerned with

placing s where it is required by other components which use the service. The agent $Prov$ thus receives the location x at which the service is required and sends s to that location. Since there may be more than one client of the service, the agent $Prov$ is defined to be a replicated process (!) which will repeatedly send out the service reference each time a location is received.

The declaration of a required service, **require** r , is modelled by the agent $Req(r, l)$ which is accessed by the Darwin name r and which manages the location l at which the service is required. Again, Darwin is not concerned with how a client component uses a service, it must ensure that a reference to the service is placed at some location in the client component. The Req agent receives the access name to a $Prov$ agent and sends the location l to that agent. A requirement in Darwin may only be bound to a single service and so the agent Req sends out the location l precisely once as shown below:

$$Req(r, l) \stackrel{\text{def}}{=} r(y).\bar{y}l$$

The binding construct in Darwin is modelled by the $Bind$ agent which simply sends the access name of the $Prov$ agent to the Req agent.

$$Bind \stackrel{\text{def}}{=} \bar{r}p$$

3.2 Components

Agents or processes in the π -calculus cannot be directly named, instead agents are accessed by named channels. Although Darwin names instances of components, these names are only used to qualify the names of the service they provide or require. This is illustrated by translating the simple non-hierarchical Darwin configuration of Figure 2 into the π -calculus.

Each primitive component is represented by an agent which is a composition of the $Prov$ and Req agents which manage its service requirements and provisions and the agents which define its behavior. A primitive component is simply a component which has no Darwin defined substructure of components. The Server component type of Figure 2 is represented by the π -calculus agent:

$$Server(p) \stackrel{\text{def}}{=} (vs)(Prov(p, s) \mid Server'(s))$$

in which $Server'$ represents the user implemented behavior of the $Server$ component which perform the services s . Similarly, the $Client$ component type is represented by the agent:

```

component Server {
  provide p;
}

component Client {
  require r;
}

component System {
  inst
    A:Client;
    B:Server;
  bind
    A.r -- B.p
}

```

Figure 2: Client Server system.

$$Client(r) \stackrel{\text{def}}{=} (\nu l)(Req(r, l) \mid Client'(l))$$

Note that the scope of the service name s is local to the *Server* agent and similarly, the name of the place at which the service is required l is also local to the *Client*. As can be seen in the following, binding extends the scope of these names.

3.3 Configurations

The configuration of the System component of Figure 2 is represented in π -calculus by the parallel composition of a *Client*, *Server* and *Bind* agent:

$$System \stackrel{\text{def}}{=} (\nu r_A, p_B)(Client(r_A) \mid Server(p_B) \mid Bind(r_A, p_B))$$

The instance names A and B in Figure 2 are used only to qualify and thus rename the requirement r and provision p of the *Client* and *Server* component types. The expression above is a precise translation of the Darwin configuration of Figure 2. To demonstrate that the model is correct, it must be shown that the client instance A will get the service reference provided by the server B when the configuration is elaborated. Substituting the definitions for *Client* and *Server* and dropping the quoted user defined behavior agents since they play no part in the binding process the π -calculus description of Figure 2 becomes:

$$(\nu r_A, p_B)((\nu l)Req(r_A, l) \mid (\nu s)Prov(p_B, s) \mid Bind(r_A, p_B))$$

$$\begin{aligned}
&\rightarrow (vp_B)((vl)\overline{p_B}l \mid (vs)Prov(p_B, s)) \\
&\rightarrow (vl, s, p_B)(\overline{ls} \mid Prov(p_B, s))
\end{aligned}$$

The expression describing the client server system thus reduces to an expression which sends the service s to the required location l in parallel with the $Prov$ agent and of course $Server'$ and $Client'$. Before the client can use the service it must perform an input action. A possible definition for $Client'$ would be:

$$Client'(l) \stackrel{\text{def}}{=} l(x).Client''$$

The system: $(vl, s, p_B)(\overline{ls} \mid Prov(p_B, s) \mid Client'(l) \mid Server'(s))$
then reduces to: $\rightarrow (vs, p_B)Prov(p_B, s) \mid Client''(s/x) \mid Server'(s)$

which is the desired result of an instance of the server component executing in parallel with a client component in which every occurrence of the local name x has been replaced with the name s , the reference to the required service. In practice, a Darwin implementation can compute the number of requirements bound to a provision so that the number of replicas of the agent $Prov$ is known. Consequently, the configuration process can terminate and the resources it uses can be recovered. It should be noted that the model described permits binding and instantiation to proceed concurrently. Components which try to use a service will be blocked until they are bound to that service (i.e. they must input the service reference as in $Client'$).

3.4 Discussion

The representation of reconfigurable systems in π -calculus illustrated in this paper by the Darwin language presents some shortcomings concerned to legibility, context independency and reconfigurability.

- **Legibility**

Here we use the term legibility not only regarding a good specification representation, but also the structural architectural materialization at specification level. As we know, one of the most important issues in architectural formalization is to enable analysis at architectural level. However, if we look only at the π -calculus representation of an architecture, we cannot distinguish any structural information regarding that architecture. We just observe lots of agents without any architectural information. This problem is caused by the lack of syntactic operators of π -calculus to represent architectural structures. For example, we cannot represent a component in π -calculus, instead we only represent its ports by means of individual agents. In Darwin representation, we use three agents to represent each connection.

- **Context Independency**

In the configuration paradigm components may be implemented and tested independently of the rest of the system in which they will form a part. These components must be reused in different contexts without performing any modification in their internal operations. However,

it is not possible to reuse the π -calculus specification of the agents that represents this component. The main problem is the dependency among ports in the π -calculus representation. In order to perform communication between two π -calculus agents they must synchronize their actions in ports with the same names and opposite directions. In other words, they must have a shared channel between each connected port of each component. This π -calculus feature limits the representation of context independent agents.

- **Reconfigurability**

A basic and the most important operation on reconfigurable systems is to change the connections among components. At architectural level we have only to change the ports binds. But it is not easy to perform this operation at specification level. If we perform a change in any bind at architectural level we have to change all the related agents that represent this bind. The π -calculus is not equipped with syntactic operators that allow the manipulation of pattern of synchronization on channels.

4 The $\mathcal{R}\pi$ -calculus

In order to support reconfigurability in π -calculus components' specification we propose an extension of π , named $\mathcal{R}\pi$, which deals with all π restrictions presented in the previous section related to reconfigurable specifications. First, we have to deal with the synchronization discipline imposed by π , in which the communication may occur if and only if the actions are equal and observable. For example:

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

The communication between the channels a and $\bar{a}$ is possible due to the fact that these ports represent an action and a coaction on the same channel, i.e., the synchronization of the two processes depends on the relation of equality on channel names.

Now suppose that the channel names of the processes are a and b , we have the following system:

$$a(x).P \mid \bar{b}y.Q$$

That system is deadlocked because there are no possible transitions by the π semantics since we do not have any synchronizable channels between P and Q .

In $\mathcal{R}\pi$ we relax this discipline allowing the synchronization of processes on different actions, i.e., we do not impose a synchronization discipline based on the relation of equality of names, but we use an equivalence relation on names, termed correlation, in such a way that two related names are considered equal. For example:

$$a(x).P \mid \bar{b}y.Q \xrightarrow{[a \diamond b]} P\{y/x\} \mid Q$$

The transition above defines that a and b are correlated (represented by the symbol $\diamond$), i.e., it can synchronize on this actions. In that way the behavior of the system is the same as in π , with the

advantage that we can create processes entirely context independent, in which the synchronization discipline is explicitly defined.

A clear problem of that approach is the vulnerability to collision of names, in which we can produce a system with processes having the same channel names, so that we are not able to identify by means of correlations which names we are referring to in a communication. To exemplify, consider the following specification, where the process R is reused from another specification:

$$[a \diamond b](a(x).P \mid \bar{b}y.Q \mid a(w).c(t).R)$$

In that system there is a conflict concerning the interaction that can occur. Considering the correlation $a \diamond b$, we can have communication between P and Q , through the channels a and b , or between R and Q , through the channels a and b . To deal with this issue, we formulate the idea of modularization of specification so that we are able to separate pieces of reusable specifications. In this sense, the notion of components is inserted into $\mathcal{R}\pi$. By means of components, that are the smallest portions of specification reuse in $\mathcal{R}\pi$, we can create context independent specifications, which can be related by correlations. In the following specification we have the component representation of the previous specification:

$$\begin{array}{c} C_1[a(x).P] \mid C_2[\bar{b}y.Q] \mid C_3[a(w).c(t).R] \xrightarrow{[C_1.a \diamond C_2.b]} \\ C_1[P\{y/x\}] \mid C_2[Q] \mid C_3[a(w).c(t).R] \end{array}$$

Now we can identify the components C_1, C_2 , and C_3 . Each component encapsulates processes whose channel names are entirely context free. The correlation $[C_1.a \diamond C_2.b]$ relates the channels a from the component C_1 and the channel b from the component C_2 .

By relaxing the synchronization discipline by means of correlations on channel names and inserting the concept of components on formal specifications, the $\mathcal{R}\pi$ -calculus seems really suitable for modelling reconfigurable component-based systems. Following we present in details the syntax and semantics of $\mathcal{R}\pi$.

4.1 Syntax

As an extension of π -calculus, the syntax of $\mathcal{R}\pi$ is defined from π by adding a few new concepts. Such new concepts are related to the notion of reconfigurability, that is the main goal of $\mathcal{R}\pi$. First, the concept of component should be defined. A component is a context independent labelled container for π processes. Each component may be composed by one or more π processes, and represents the smallest unit of reconfigurability and usability of $\mathcal{R}\pi$.

As previously presented in the examples, the communication among components in $\mathcal{R}\pi$ is controlled by the synchronization of processes on related channels in different components, in which such relation, termed correlation, takes into account the names of components and the names of the channels.

Like π -calculus, $\mathcal{R}\pi$ postulates an infinite set $\mathcal{N}$ of *names* ranged over by $a, b, c, \dots, x, y, z$ (except p). Let τ be a distinguished element such that $\mathcal{N} \cap \{\tau\} = \emptyset$. These names represent the communication ports.

Let $\mathcal{C}$ be a set of all component names, ranged over by C_1, C_2, C_3 . And let $\mathcal{P}$ be the set such that $\mathcal{P} = \mathcal{C} \times \mathcal{N}$, which elements range over by p_1, p_2, p_3 . For convenience, we write $C_1.a$ to represent the element $p = (C_1, a)$.

We also define ψ as the set of all equivalence relations over $\mathcal{P}$. We write $[p_1 \diamond p_2]$ to represent the smallest equivalence relation relating p_1 and p_2 , where $p_1, p_2 \in \mathcal{P}$. Also for convenience, we use 1 to describe a reflexive relation. As a consequence, the relation $[p_1 \diamond p_1]$ is the same as $[p_2 \diamond p_2]$.

Definition 1 (Actions)

Actions are input, output, ranged over by γ and correlation ranged over by ψ , defined by the following grammar:

$$\begin{array}{ll} \gamma & ::= C_1.a(x) \quad (\text{Input}) \\ & \quad | C_2.\bar{a}y \quad (\text{Output}) \\ \psi & ::= [p_1 \diamond p_2] \quad (\text{Correlation}) \end{array}$$

Let $\mathcal{A}$ be the set of all actions. We use α as meta variable to range over actions.

1. $a(x)$ is called an *input prefix*. A name a is thought of as an input port of a process which contains it.
2. $\bar{a}y$ is called an *output prefix*. A name a is thought of as an output port of a process which contains it.
3. $[p_1 \diamond p_2]$ is called a *correlation*. An element p_1 is correlated to p_2 , in other words, suppose $p_1 = C_1.a$ and $p_2 = C_2.b$, $[p_1 \diamond p_2]$ is a relation represented by the following set: $\{(C_1.a, C_2.b), (C_2.b, C_1.a)\} \cup 1$, where 1 represents the identity relation.

Definition 2 (Agents)

The agents ranged over by $P, Q, R, \dots$, are defined by

$$\begin{array}{ll} P & ::= \emptyset \quad (\text{Inaction}) \\ & \quad | \alpha.P \quad (\text{Prefix}) \\ & \quad | Q + R \quad (\text{Summation}) \\ & \quad | Q \mid R \quad (\text{Composition}) \\ & \quad | (\nu x)Q \quad (\text{Restriction}) \\ & \quad | A(x) \quad (\text{Identification}) \end{array}$$

The agents (or processes) are encapsulated into components and have the same behavior as defined in π .

Definition 3 (Components)

The components ranged over by C, C_1, C_2 , are defined by

$$\begin{array}{ll}
C ::= \emptyset & (\text{Empty Component}) \\
| \psi.C_1[P] & (\text{Prefixed Component}) \\
| C_1 | C_2 & (\text{Component Composition})
\end{array}$$

Components are containers for π agents and the intra-component communication is controlled by means of correlations, whereas all the communication that takes place inside a component, i.e., among processes that form this component, are regulated by the usual π -calculus rules, including structural congruence, reduction rules, and so on.

We define components' free (fn) and bound (bn) names as follows:

$$\begin{array}{ll}
fn(C[\emptyset]) & \xrightarrow{\text{def}} \emptyset \\
fn(C[P]) & \xrightarrow{\text{def}} fn(P) \\
fn(C[P | Q]) & \xrightarrow{\text{def}} fn(P) \cup fn(Q) \\
fn(C[P + Q]) & \xrightarrow{\text{def}} fn(P) \cup fn(Q)
\end{array}$$

Definition 4 (Substitution)

A substitution is a mapping σ from $\mathcal{N}$ to $\mathcal{N}$. Suppose $\sigma(x_i) = y_i$, for all $1 \leq i \leq n$ and $\sigma(x) = x$ for all other names x , we write $\{y_1/x_1, \dots, y_n/x_n\}$. $C_1[P]\sigma$ denotes the expression obtained from $C_1[P]$ by simultaneously substituting $\sigma(x)$ for each occurrence of x in P for each x , with changes of bound names where necessary, as usual.

The structural congruence equates agents that are not distinguished by any semantic reason.

Definition 5 (Structural Congruence)

The structural congruence, represented by $\equiv$, is the smallest congruence relation satisfying the following laws:

1. $C_1[a(x).\bar{b}x] \equiv C_2[a(y).\bar{b}y]$
 $C_1[a(x).\bar{b}x] \equiv C_2[c(x).\bar{d}x]$ if $[C_1.a \diamond C_2.c, C_1.b \diamond C_2.d]$
2. $C_1 | 0 \equiv C_1$
 $C_1 | C_2 \equiv C_2 | C_1$
 $C_1 | (C_2 | C_3) \equiv (C_1 | C_2) | C_3$
3. $C_1[P] \equiv C_2[Q] \Leftrightarrow P \equiv Q$
4. $C_1[P | Q] \equiv C_1[P] | C_1[Q]$
5. $C_1[P + Q] \equiv C_1[P] + C_1[Q]$
6. $C[0] \equiv 0$

STR	$\frac{C_1[P'] \equiv C_1[P], C_1[P] \xrightarrow{C_1 \cdot \alpha} C_1[Q], C_1[Q] \equiv C_1[Q']}{C_1[P'] \xrightarrow{C_1 \cdot \alpha} C_1[Q']}$	(Struct)
PREF	$\frac{-}{C_1[\alpha.P] \xrightarrow{C_1 \cdot \alpha} C_1[P]}$	(Prefixing)
SUM	$\frac{C_1[P] \xrightarrow{C_1 \cdot \alpha} C_1[P']}{C_1[P] + C_2[Q] \xrightarrow{C_1 \cdot \alpha} C_1[P']}$	(Summation)
PAR	$\frac{C_1[P] \xrightarrow{C_1 \cdot \alpha} C_1[P'], \mathbf{fn}(\alpha) \cap \mathbf{fn}(Q) = \emptyset}{C_1[P] \mid C_2[Q] \xrightarrow{\alpha} C_1[P'] \mid C_2[Q]}$	(Composition)
COMM	$\frac{C_1[P] \xrightarrow{C_1 \cdot a(x)} C_1[P'], C_2[Q] \xrightarrow{C_2 \cdot \bar{b}y} C_2[Q'], C_1.a \diamond C_2.b}{C_1[P] \mid C_2[Q] \xrightarrow{[C_1 \cdot a \diamond C_2 \cdot b]} C_1[P'\{y/x\}] \mid C_2[Q']}$	(Communication)

Table 1: Labelled Transition Semantics.

We have just presented the rules related to components, all the rules related to processes that are encapsulated into components are already defined in π . In these rules we define the components satisfying the abelian monoid laws for summation and composition. In addition, in rule 1, we define an extended notion of alpha-conversion for components. In that rule, we can observe that components made up of agent variants of alpha-conversion, i.e., in the choice of bound names, are considered the same, and components whose free names are correlated and bound names are equal are also considered the same.

4.2 Operational Semantics

The standard way to define an operational semantics to a process algebra is through a labelled transition system. The operational semantics to $\mathcal{R}\pi$ is defined in the following way:

Definition 6 (Labelled Transition Semantics)

The family of transitions $\xrightarrow{\alpha} (\subset \mathcal{C} \times \mathcal{C})$, for $\alpha \in \mathcal{A}$, is the smallest family that satisfies the laws in Table 1.

The semantics of $\mathcal{R}\pi$ defines the communication among components. All the interactions are related to components, in which synchronization actions are defined by means of correlations, in which pairs of elements indicate which names are equivalent. The synchronization of process actions inside the components are regulated by the usual π -calculus rules.

Basically, the rules presented in the Table 1 have the same structure as the corresponding rules in π applied to components. In these rules we present the transition system based on compound

names, represented by the name of the component plus the name of the channel and on correlations between these compound names.

The most expressive rule of $\mathcal{R}\pi$ is the [COMM] rule, that represents the synchronization of components on different channel names related by a correlation. With that rule we can build context independent components and reconfigurable systems, in which the synchronization discipline of components is set by the system itself.

5 Compatibility Check

In addition to introducing equivalence between channel names in communicating components by means of correlations, we have to guarantee the behavioral compatibility of these channels, i.e., consider the components $C_1[a(x).P]$ and $C_2[\bar{b}y.Q]$ and the relation $[C_1.a \diamond C_2.b]$ between them, we have to guarantee that the behavior of the communication between $C_1.a$ and $C_2.b$ is compatible, i.e., communication can take place. In the example, we can say that the components C_1 and C_2 are compatible.

The definition of component compatibility lies in the notion of compatibility among processes that compose the components considering the correlations, i.e., we have to test the compatibility of each pair of processes that compose the component observing the equivalences defined between their names. We can say that two components are compatible if they can be composed in parallel without a deadlock, i.e., without behaving like the inaction.

Additionally, the notion of compatibility can guarantee that when we chose a component to reuse in another system the behavior of this component is compatible with the system as a whole, i.e., the system will work properly after the addition of the new component. By means of compatibility check we have a behavioral-centric approach to select software components to a system.

The compatibility relation formalizes a way to recognize when two components specified in $\mathcal{R}\pi$ are compatible. We say that two components are compatible (1) they can engage in at least one transition, and (2) any complementary transitions will lead to compatible components.

Firstly, for two components to engage in one transition they have to synchronize in their actions. Thus, we define synchronizable components in the following way:

Definition 7 (Synchronizable Components)

Components $C_1[P]$ and $C_2[Q]$ are synchronizable if $\exists C_1[P']$ and $C_2[Q']$ such that,

$$C_1[P] \xrightarrow{C_1.\bar{a}x} C_1[P'], C_2[Q] \xrightarrow{C_2.b(y)} C_2[Q'] \text{ and } [C_1.a \diamond C_2.b], \text{ or} \\ C_2[Q] \xrightarrow{C_2.\bar{a}x} C_2[Q'], C_1[P] \xrightarrow{C_1.b(y)} C_1[P'] \text{ and } [C_1.a \diamond C_2.b]$$

Secondly, we need that components, after engaging in a transition, lead to compatible components. In that way, we define following the notion of semi-compatibility:

Definition 8 (Semi-Compatibility Relation)

Components $C_1[P]$ and $C_2[Q]$ are related by a semi-compatibility, written $C_1[P] \mathcal{R} C_2[Q]$, if

1. C_1 and C_2 are synchronizable.
2. if $C_1[P] \xrightarrow{C_1.\bar{a}x} C_1[P'], C_2[Q] \xrightarrow{C_2.b(y)} C_2[Q']$ and $[C_1.a \diamond C_2.b]$ then $C_1[P'] \mathcal{R} C_2[Q'\{b/a\}]$

Definition 9 (Compatibility Relation)

The relation $\mathcal{R}$ is a compatibility if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are semi-compatibilities. We define $\simeq$ as the least compatibility on components.

We say that components are compatible if there are a correlation between them and their compatibility relation is preserved under substitutions as defined as follows:

Definition 10 (Correlated Components)

The components C_1 and C_2 are correlated, written $\div$, if $\exists a(x).P \in C_1$ and $by.Q \in C_2$, such that $C_1.a \diamond C_2.b$.

Definition 11 (Compatible Components)

The components C_1 and C_2 are compatible, written $C_1 \approx C_2$, if

1. $C_1 \div C_2$, and
2. $C_1\sigma \simeq C_2\sigma$ for all substitutions σ .

The reusability of a component in a given system is tested by means of compatibility of correlated components. The following definitions deal with this issue:

Definition 12 (Reusable Component)

Let C_1 be a component and Ω a specification, C_1 is reusable in Ω if $\exists C_2 \in \Omega$ such that $C_1 \approx C_2$.

6 Example

To illustrate the application of $\mathcal{R}\pi$ in specifying reconfigurable systems, we will use the Group Practice of Doctor (GPD) application, adapted from [4]. This application has three types of processes: *receptionist*, *patient* and *doctor*. The intended semantics is as follows. The *patient* arrives and says his name and illness to the *receptionist*. On the other hand, *doctors* who are capable of receiving patients inform the receptionist of their availability. The receptionist informs the available doctor the names and illnesses of his patients. The patient is given a prescription by the doctor. After that, the patient leaves and the doctor becomes free to receive another patient. The Figure 3 presents the π -calculus modelling of this application [2].

To reason about the system properties we have to compose the agent definitions using the parallelism operator of π -calculus. The resulting system is as follows:

$$RECEPTIONIST(next, talk) \mid DOCTOR(next) \\ \mid PATIENT(n, talk)$$

$$talk(n, ill, reply).next(answer).\overline{answer}(n, ill, reply) \mid \\ (v\ answer) next(answer).answer(n, ill, reply).reply(diagnose(ill)) \mid \\ (v\ reply) talk(n, rand(), reply).reply(prescription).0$$

$$\begin{aligned}
RECEPTIONIST(next, talk) &\stackrel{\text{def}}{=} talk(n, ill, reply). \\
&\quad next(answer).\overline{answer}(n, ill, reply). \\
&\quad RECEPTIONIST(next, talk) \\
DOCTOR(next) &\stackrel{\text{def}}{=} (v \text{ answer}) \\
&\quad \overline{next}(answer).answer(n, ill, reply). \\
&\quad \overline{reply}(diagnose(ill)). \\
&\quad DOCTOR(next) \\
PATIENT(n, talk) &\stackrel{\text{def}}{=} (v \text{ reply}) \\
&\quad \overline{talk}(n, rand(), reply). \\
&\quad reply(prescription).0 \\
SYSTEM &\stackrel{\text{def}}{=} RECEPTIONIST(next, talk) \mid \\
&\quad DOCTOR(next) \mid PATIENT(n, talk)
\end{aligned}$$

Figure 3: The Group Practice Doctor π -calculus Specification

We can observe that components specified in π -calculus have a global memory of the system code source, i.e., their channels are built based on the channels of the other processes that make up the system. Hence, we cannot build context independent formal component specifications, since the π -calculus as well as the other process algebras, follow a restricted synchronization discipline based on equality of name channels. Using $\mathcal{R}\pi$ to relax this property we can solve this problem and build really context independent formal component specifications. To exemplify, suppose we get the follow specification of the component *PATIENT* from another system.

$$(v \text{ response}) \overline{ask}(n, rand(), response).response(receipt).0$$

Following the idea of reconfigurability, the *PATIENT* is built as a context independent component, i.e., we should not worry about the external environment to build the specification. In this case, the names of the channels are independent of any system context. At architectural level, to reconfigure the previous system using this new component, we have just to connect the appropriate ports. However, this fact cannot be mimic at specification level, i.e., using π -calculus we cannot reconfigure the system. However, using $\mathcal{R}\pi$ we can build the following specification:

$$\begin{aligned}
&[C_1.talk \diamond C_3.ask, C_1.next \diamond C_2.next, C_1.answer \diamond C_2.answer, \\
&C_2.reply \diamond C_3.response] \\
&(C_1[talk(n, ill, reply).next(answer).\overline{answer}(n, ill, reply)] \mid \\
&C_2[(v \text{ answer}) \overline{next}(answer).answer(n, ill, reply). \\
&\overline{reply}(diagnose(ill))] \mid \\
&C_3[(v \text{ response}) \overline{ask}(n, rand(), response).response(receipt).0])
\end{aligned}$$

In the $\mathcal{R}\pi$ specification, all the (re)configurations are made using the correlations. In this sense, the channel names are connected at configuration time, and the components are context independent, what makes $\mathcal{R}\pi$ a calculus that captures the notion of reconfigurability also at formal specification level. In addition, we can perceive the application structure mapped in the formal specification, i.e., we can identify the components and the connections.

Using the $\mathcal{R}\pi$ system specification we can test the compatibility of *PATIENT* in the system, i.e., if the components C_1 and C_2 are compatible or $C_1 \approx C_2$.

By the Definition 11, in order to prove this property we have to show:

1. $C_1 \doteq C_2$
2. $C_1\sigma \simeq C_2\sigma$ by all substitution σ

Proof:

- In the system specification we have $[C_1.next \diamond C_2.next, C_1.answer \diamond C_2.answer]$, thus, by the Definition 10, C_1 and C_2 are compatible;
- $C_1\sigma \simeq C_2\sigma$ by all substitution σ , since $fn(C_1) \cap fn(C_2) = \emptyset$;

7 Related Work

In terms of the formal basis, we propose the use of a process algebra to specify reconfigurable systems by means of components. A number of other works have been using process algebras to deal with the notion of containers of processes in the same way as us. The M-pi calculus [3], for example, extends π -calculus with the notion of modules and object, in which semantics allows operations like **join** and **sep** to deal with module and interface interactions.

In $\mathcal{R}\pi$ we propose a π -calculus extension to deal with the notion of components at the specification level, in which components are the least unit of specification and reconfiguration. In addition, $\mathcal{R}\pi$ deals with the restrictive synchronization discipline of π in which only channels with the same name can synchronize in an action, in $\mathcal{R}\pi$ components with different channel names, related by a correlation, can also synchronize in actions.

The reuse at specification level is considered in [8], but in other terms than our proposal. In that work, the reuse problems of selection and adaptation were dealt with at specification level, on the other hand, in our work we propose to produce reusable specifications, in which the problems of reuse can be formally solved.

8 Conclusions

The use of formal models for specifying composable systems is recommended to facilitate the selection of components and the test of the system as a whole before its implementation. However, we argue that it is not enough to formalize each component of this system. We have to be able to use these specifications in the context of other systems, moreover we must be able to verify the reusability of components and allow the reconfiguration of the systems.

Due to their compositionality, the process algebras, like π -calculus have been used for modelling composable systems. However, to reconfigure π -calculus specifications we have to adapt the processes' channel names according to the system in which it will be inserted. Thus, the reconfiguration of a complex system tends to be a very hard task.

In this paper we proposed the idea of reconfiguration at the specification level. We argue that the construction of context independent formal models may affectively help the development of complex composable systems.

To deal with this new concept, we propose the $\mathcal{R}\pi$ -calculus. $\mathcal{R}\pi$ is a π -calculus extension that uses the concept of component as unit of specification and whose semantics allows the reconfiguration of these components in any system without the manipulation of channel names. We demonstrate the benefits of $\mathcal{R}\pi$ with of a simple example. However, we are confident that similar results may be achieved when modelling more complex applications.

We also developed the π CL language [9], that is a reconfiguration language whose underlying semantics is based on $\mathcal{R}\pi$. Using π CL we are able to build reconfigurations in an abstract way and test system behavior by means of $\mathcal{R}\pi$ specification, which is automatically reconfigured to reflect the system structure.

References

- [1] Canal, C. and Fuentes, L. and others. Extending CORBA Interfaces with Protocols. *The Computer Journal*, April 2001.
- [2] Eisenbach, Susan and Paterson, Ross. π -Calculus Semantics for the Concurrent Configuration Language Darwin. In *Hawaii International Conference on Systems Science*, 1993.
- [3] Kanezashi, Fumiaki and Togashi, Atsushi. M-pi Calculus: A Calculus for Module Description. *International Journal of Computer & Information Science (IJCIS)*, 2001.
- [4] Magee, J. and Dulay, N. and Kramer, J. Structuring Parallel and Distributed Programs. In *Proceedings of The IEEE International Workshop on Configuring Distributed Systems*, March 1992.
- [5] Milner, R. *Communication and Concurrency*. Prentice Hall, 1989.
- [6] Milner, R. and Parrow, J. and Walker, D. A Calculus of Mobile Processes. In *Information and Computation*, 1992.
- [7] Oberndorf, P. and Browsord, L. Are You Ready for COTS? Technical report, Software Engineering Institute, 1997.
- [8] Penix, John and Alexander, Perry. Toward Automated Component Adaptation. In *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering*, 1997.
- [9] Souza, Cidcley T. de and Cunha, Paulo R. F. A Behavioral-Centric Approach for Specifying Component-Based Systems. *XV International Conference on Systems Engineering (ICSEng2002)*, 2002.

- [10] Tran, V. and Liu, D. B. A Risk-Mitigating Model for The Development of Reliable and Maintainable Large-Scale Commercial-Off-The-Shelf Integrated Software Systems. In *Proceedings of the 1997 Annual Reliability and Maintainability Symposium*, 1997.