Software Architecture Specification with π-calculus [*]

Carlos Canal Ernesto Pimentel José M. Troya
Dept. de Lenguajes y Ciencias de la Computación
University of 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 levels of the development process. In this paper we focus in the level of design called Software Architecture. At this level, the system is described as a collection of interrelated components. Process algebras, such as Milner's π-calculus seem to be a good candidate for the specification of system's components. However, the relations of bisimilarity established for the π-calculus are not well suited for this level of specification, where a less strict relation, which we call compatibility, would be preferable. Thus, we introduce a relation of process compatibility in the context of the π-calculus which formalizes the intuitive notion of compatibility among processes. The suitability of our approach is shown by means of a case study.

1 Introduction

The success on the application of formal methods to the development of software depends on our ability in finding models and formalisms adequate for each of the levels of the development process. To this effect, process algebras are widely accepted for the specification of software systems. The systems so specified can be checked for equivalence, deadlock freedom, and other interesting properties.

The increasing complexity of software systems led recently to the adoption of the term Software Architecture (SA), 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 [5]. SA focuses in those aspects of design and development which cannot be suitably treated inside the modules which compose the system [13]. Among them, are included those which derive from the structure of the system, i.e. from the way in which its different components are combined.

[*]This work was funded in part by the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) under grant TIC94-0930-C02-01.

Despite of the growing importance of architectural aspects in software development, their descriptions are usually limited to the informal use of certain idioms [12], such as client/server architecture, layered architecture, etc. These descriptions lack of a precise meaning, which limit its utility to a great extent [1], preventing any attempt of analysis of the properties of the systems so described. The use of formal methods could help us to avoid these limitations. Formal specifications have a precise meaning derived from the semantics of the notation used and they admit several forms of reasoning.

In particular, we propose the use of the π-calculus [9] for the specification of software architectures. The π-calculus can express directly the mobility, making easier the specification of dynamic systems. Besides, its formal basis allows the analysis of the specifications and also the development of automated verification tools [15].

One of the properties that we may analyze at this level is composability, i. e. the capability of the system of being constructed by combining its components as indicated in its architecture. Composability can be checked by determining whether the components of the system are compatible or not. Furthermore, in order to enhance reusability, we should be able to check if a certain existing component can be used in a new system where a similar function is required. Again the intuitive notion of compatibility arises.

Compatibility can be determined by composing in parallel the elements under checking, and analyzing the resulting system for deadlocks. However, this analysis would be impractical for complex systems. Instead of that, we propose the use of explicit interface specifications for each attachment between elements in the system (what we will call roles and ports) and then checking individually the compatibility of each of the attachments. This reduces the complexity of the analysis in a great extent.

Following different proposals in this field [4, 14], we specify a software system as a parallel composition of several components and connectors, where the components model data storage and computational units and the connectors represent the mechanisms by means of which the components interact. The architecture of a system indicates the attachments of components and connectors that build it from its elemental units. Components are specified as a set of *ports*, which describe the behavior of a component in relation to the connectors it is attached to. On the other hand, connectors are characterized by a set of *roles* which describe the behavioral requirements that a connector imposes on the components attached to it. Then, the composability of the system can be checked by local analysis of each of the port-role attachments. As we usually want to attach to a role a port that matches only partially, bisimilarity analysis is not suitable, so we have defined a relation of compatibility in the context of π-calculus.

2 Related work

In the last years several proposals related to the specification of software architectures have been presented. A common pattern in all of them is that they are compositional. Some, like Darwin [8] or Rapide [7], consider that systems are built from components, while others, like Wright [4] or UniCon [14], also includes connectors as first-order elements of the notation. Because of the variety and complexity of the interaction mechanisms used in present software systems, we have adopted the second alternative. If we focus in formal methods, several papers [2, 6] propose the use of CSP, Z or the Chemical Abstract Machine for architecture specification. In [8] the π-calculus is used for defining the semantics of

Darwin, but the modeling of components' interaction mechanisms is not considered.

With regard to compatibility, our proposal is similar to [3], which uses CSP to determine protocol compatibility of ports and roles, although the possibility of direct expression of mobility in π-calculus makes easier the specification of ports and roles. In fact, as it is stated in [8], formalisms like CCS or CSP do not seem the most appropriate for the description of evolving or dynamic structures.

3 The π-calculus

The π-calculus [9] is a process algebra derived from CCS. It is specially suited for the description and analysis of dynamic systems.

A system is specified in the π-calculus as a collection of processes or agents which interact by means of links or names (we denote by $\mathcal{N}$ the set of names). The restriction of the scope of a name allows to establish links that are private to a group of processes. Processes are built from names and operators by following the following syntax:

$$P ::= \mathbf{0} \mid \alpha.P, \alpha \in \mathcal{A} \mid P_1 + P_2 \mid P_1|P_2 \mid (x)P,\ x \in \mathcal{N} \mid [x = z]P \mid A(\tilde{x})$$

where $\tilde{x}$ is a sequence of elements in $\mathcal{N}$, $A(\tilde{x})$ is a procedure call, and the set of atomic actions is given by $\mathcal{A} = \{x(y), \bar{x}y :\ x, y \in \mathcal{N}\} \cup \{\tau\}$. The process $\mathbf{0}$ prepresents the inactive process. A prefix action $\alpha.P$ performs an action α, and then behaves like P. Silent transitions (given by τ) model internal actions and, when used with the alternative operator, they enable local choices. An action $\bar{x}y$ is called *negative prefix*, and it represents the sending of the name y along the channel x. Similarly, an action $x(y)$ is called *positive prefix*, and corresponds to the complementary action. A summation of two processes, $P_1 + P_2$, behaves like P_1 or P_2 (non-deterministically). It can be used to express either local or global choices. In the parallel composition of two processes, they may evolve independently or synchonize, matching a positive prefix in P_1 (resp. P_2) with the corresponding negative prefix in P_2 (resp. P_1). This synchronization will be externally observed as a silent transition τ. The restriction operator $(x)P$ allows the hiding of the name x.

Basically, communication in the π-calculus is performed matching synchronously a negative prefix $\bar{x}z$ with the corresponding $x(y)$, as stated in the reduction rule:

$$(\cdots + \bar{x}z.P_1 + \cdots) \mid (\cdots + x(y).P_2 + \cdots) \xrightarrow{\tau} P_1|P_2\{z/y\}$$

4 Process Compability in the π-calculus

Compatibility among processes cannot be determined studying the bisimilarity of their specifications, as we usually want to attach processes that match only partially. Besides, the use of requested and real behavioral specifications determines the asymmetry of the relation. Through this section we will use the term *role* for referring to the requirements of a certain process (A) in relation to a certain attachment, while *port* will refer to the real behavior of another process (B) in relation to the same attachment. Thus, A and B can be attached or composed in parallel, if the port of A is compatible with the role of B.

A formal characterization of compatibility is given in Definition 4.2 below. Roughly, we may say that a port is compatible with a role if i) they can engage in at least one

$R_1 = a + b$	constrains the port to be able to accept an event a and also to be able to accept b. After that it must become an inaction.
$R_2 = a + t.b$	any compatible port must be able to accept a, but it may also accept b. τ transitions in roles indicate decisions of the port.
$R_3 = \tau.a + \tau.b$	any compatible port must accept a and/or b. The choice is local to the port.
$P_1 = a + b$	the port is able to accept a or b and then become inactive. The choice is not made locally by the port.
$P_2 = a + \tau.b$	the port is able to accept an event a if presented, but it may also decide by itself to engage in accepting b. Silent transitions are used in ports to model local choices.
$P_3 = \tau.a + \tau.b$	the port decides locally to engage in a or b.

Figure 1: Some examples of port and role specification

transition, ii) the port does not engage in transitions which are not foreseen in the role, iii) the port behaves in the way determined by any local choice of the role, and iv) when port and role engage in a certain transition the resulting processes are compatible.

Silent transitions, combined with the alternative operator, are used to indicate whether a process can make a local choice or not, and therefore they have different meanings in ports and roles. In a role, they indicate permission to decide locally whether to commit to a certain transition or not (and even to present a certain behavior or not). However, in a port they indicate a non-deterministic decision to commit to a certain transition. To illustrate this difference, Figure 1 contains some examples of roles (R_i) and ports (P_j) and explains the behavior that they specify.

In the definitions below the following shorthands are adopted:

$$\begin{array}{lll} \Rightarrow & \text{stands for} & (\xrightarrow{\tau})^* \text{ and} \\ \overset{\alpha}{\Rightarrow} & \text{stands for} & \Rightarrow\xrightarrow{\alpha} \quad \text{where } \alpha \neq \tau \end{array}$$

Definition 4.1 (Related processes) *Two processes, represented respectively by a port P and a role R, are related if*

i. $P \Rightarrow \mathbf{0} \,\wedge\, R \Rightarrow \mathbf{0}$*, or*

ii. $\exists\, \alpha, P', R'.\ \ P \overset{\alpha}{\Rightarrow} P' \,\wedge\, R \overset{\alpha}{\Rightarrow} R'$

This definition ensures that, provided they do not progress to inactions, the processes represented by the port and the role can engage at least in a common transition. This is a necessary but not sufficient condition for compatibility, preventing to consider compatible not related processes such as those represented by $P = a + b$ and $R = c + d$, which would deadlock when composed in parallel.

Definition 4.2 (Relation of compatibility) *A relation $\mathcal{C}$ on processes is a relation of compatibility if $P\ \mathcal{C}\ R$ implies, for every substitution σ:*

i. $P\sigma$ *and* $R\sigma$ *are related,*

ii. if $P\sigma \xrightarrow{\tau} P'$ *then* $P'\ \mathcal{C}\ R\sigma$*,* *iii. if $R\sigma \xrightarrow{\alpha} R'(\alpha \neq \tau)$ then $\exists P'$. $P\sigma \overset{\alpha}{\Rightarrow} P'$ and $P' \ \mathcal{C}\ R'$,*

iv. if $P\sigma \xrightarrow{\alpha} P'(\alpha \neq \tau)$ and $R\sigma \overset{\alpha}{\Rightarrow} R'$ then $P' \ \mathcal{C}\ R'$

The second condition ensures that any choice of the port is supported by the role. On the contrary, the third one ensures that the port is able to follow any requirement of the role. The last one ensures that common transitions lead to compatible processes.

As we will see below, the parallel composition is the way in which links are stablished among components and connectors. For this reason, we need to proof the following result claiming that a compability relation is a congruence w.r.t. parallel composition.

Proposition 4.3 *Given a compatibility relation $\mathcal{C}$, if $P_1 \ \mathcal{C}\ Q_1$ and $P_2 \ \mathcal{C}\ Q_2$ then:*

$$P_1|P_2 \ \mathcal{C}\ Q_1|Q_2$$

Proof. Let P, Q be $P_1|P_2$ and $Q_1|Q_2$, respectively. P and Q are obviously related. □

The definition of compatibility involves universal quantification over substitutions. However, if we construct a specialized transition system like the one used by Sangiorgi in [11], it is possible to develop an efficient automated verification tool for the relation.

Definition 4.4 (Compatible processes) *The processes P and R are compatible, written $P \rhd R$, if $P \ \mathcal{C}\ R$ for some compatibility relation $\mathcal{C}$.*

Returning to the examples of Figure 1, we have that P_1 is compatible with roles R_1, R_2 and R_3, because it doesn't commit by itself to any particular transition and is able to accept any local choice of the roles. However P_2 and P_3 are compatible with role R_3 only. P_2 is not compatible with role R_1 because they don't fulfill the second condition as P_2 may perform a silent transition, becoming b, and b is not compatible with R_1. Similarly P_3 is not compatible with role R_1 nor with role R_2.

Notice that there is no logical implication between compatibility and bisimilarity. Even identical processes, may be not compatible because of the different interpretations that we give to silent transitions in ports and roles.

5 Formal Architecture Specification with π-calculus

We will describe the architecture of a system as a collection of computational and data storage units interconnected in a certain way. Such a system can be specified in the π-calculus by a set of agents related through names.

```
InputPort(input) =
  input(x).
  InputPort(input)
```

```
Inverter(input,output) =
  input(x).
   ([x=TRUE]output FALSE.
   Inverter(input,output)
   +[x=FALSE]output TRUE.
   Inverter(input,output))
```

```
OutputPort(output) =
  τ.output FALSE.
    OutputPort(output)
  +
  τ.output TRUE.
    OutputPort(output)
```

Figure 2: An inverter component with its ports

InputRole(input) = $\overline{\texttt{input}}$(x). InputRole(input)	Channel(input,output) = (x)(input(x).$\overline{\texttt{output}}$ x. Channel(input,output))	OutputRole(output) = output(x). OutputRole(output)

Figure 3: Data channel connector with its roles

5.1 Component and Connector Specification

Components and connectors can be either simple or composite (i.e. a subsystem formed by several components and connectors). The specification of a composite component or connector in the π-calculus is an agent whose definition equation consists on several (simple or composite) components and connectors in parallel. We can see that these specifications are hierarchical and modular, and that the use of name restrictions ensures protection and encapsulation.

A certain component can be attached to several connectors at the same time. The set of free names of a component represents its interface, and is expressed by the parameters which appear in its agent definition equation. This interface can be partitioned into several *ports*, each of them referring to the interaction of the connector with a certain component. So, a port is an abstraction partially describing the behavior of a component as seen from a certain connector attached to it. The arguments of a port are a subset of those of the corresponding component.

The interaction with other connectors attached to the component is not explicitly expressed in the port, but if desired, may be expressed as silent transitions. However, when some of these interactions may have influence in port's behavior they must be expressed as local choices, combining the alternative operator and silent transitions.

The latter can be explained with an example. An *Inverter* component has two ports, input and output. Figure 2 contains the specification of the whole component and its ports. Observe that actions which refer to the output are not present in the input port, and that actions referring to the input are not explicit in the output port but are considered as local choices, representing a non-deterministic behavior of the component when seen from its output port.

Each of the components attached to a connector plays a certain *role* in the connection. So we will describe connectors' interfaces as a set of roles. Each role specifies the requirements that the connector imposes on the component attached. Unlike ports, roles are not partial specifications of connectors, but describe patters that components must match, as we can see in Figure 3. In this context, the attachment of a component to a connector consists on the attachment of one of the component's ports to one of the connector's roles.

The notion of observability we are interested in, only takes into account silent transitions. Therefore, we consider that a process is deadlock-free when it reaches inaction after a (possibly empty or infinite) sequence of silent transitions.

5.2 System Architecture Specification

This section illustrates the implications of port and role compatibility. First, we need to define when a set of roles actually specify the behavior of a connector.

Definition 5.1 (Correctness of a connector's roles) *Let Conn be a connector and* $\{R_i\}_i$ *its roles.* $\{R_i\}_i$ *correctly specify Conn when*

i. $fn(R_i) \cap fn(R_j) = \emptyset, \quad i \neq j$

ii. $Conn \xrightarrow{\alpha} Conn'$ *iff* $\prod_i R_i \xrightarrow{\tau}\overset{\bar{\alpha}}{\Rightarrow} \prod_i R'_i$

iii. $Conn \xrightarrow{\tau}\overset{\alpha}{\Rightarrow} Conn'$ *iff* $\prod_i R_i \xrightarrow{\bar{\alpha}} \prod_i R'_i$

and $\{R'_j\}_j$ *correctly specify Conn'.*

Intuitively this means that the roles specify completely the interface of the connector fulfilling all its requirements. The first condition ensures the connector specification is made modularly, in such a way that different roles cannot synchronize. The parallel composition of the connectors and the roles would be externally observed as a sequence of silent transitions (possibly) leading to an inaction, as states the proposition below.

Proposition 5.2 *Let Conn be a connector and* $\{R_i\}_i$ *a set of roles which correctly specify Conn. Let* v *be the free names in Conn. Then* $(v)(Conn|\prod_i R_i)$ *is deadlock-free.*

Proof. It can be directly derived from Definition 5.1. □

This property is a consequence of having a correct specification for a connector. Similar restrictions are necessary for components and ports, but the results we present below only concern to ports, and components are not directly involved. However, when a connector *Conn* is given and a port P is found compatible with some of its roles R, we will assume that free names in P not occurring in R are conveniently renamed (or hidden) to avoid collisions with free names in *Conn* and the rest of roles. In this way, we guarantee the modularity and the encapsulation of a system specification.

Theorem 5.3 *Let Conn be a connector correctly specified by roles* $R_1, R_2, \cdots, R_n$*. Let* P_i *be ports such that* $P_i \rhd R_i, for i = 1...n$*. Let* v *be the free names in Conn. Then we have that* $(v)(Conn|\prod_i P_i)$ *is deadlock-free.*

Proof. It can be proved from 4.2 and 5.1 that all traces of $(v)(Conn|\prod_i P_i)$ are also traces of $(v)(Conn|\prod_i R_i)$. If the latter is deadlock-free the first will be also deadlock-free. □

Until now, we have considered only one connector. Nevertheless, the architecture of a system is more complex. This is the aim of the following definition.

Definition 5.4 (Architecture of a system) *Consider a system formed by the parallel composition of a set of components* $\{Comp_i\}_i$ *and a set of connectors* $\{Conn_j\}_j$*. Let* $\mathcal{P}$ *be the set of port definitions of the components and* $\mathcal{R}$ *be the set of role definitions of the connectors. We define an architecture of the system as a bijective mapping* $\varphi : \mathcal{P} \longrightarrow \mathcal{R}$ *such that for every* $P \in \mathcal{P}$, $P \rhd \varphi(P)$*, and where free names in* P *not appearing in* $\varphi(P)$ *are conveniently renamed to avoid collisions with all connectors and the rest of ports.*

For simplicity we have defined the architecture of a system as a set of one-to-one attachments and so we have committed ourselves to bijective mappings. However, a more general definition can be considered using non-bijective mappings allowing, for instance, free ports and roles in the system or several ports attached to the same role. A similar result as that proved in Theorem 5.3 can be proved for a full architecture.

```
FTPUser(connect) = (ctrl)
  (τ.c̅o̅n̅n̅e̅c̅t̅ ctrl.FTPUserWaitingConnection(connect,ctrl) + τ.0)
FTPUserWaitingConnection(connect,ctrl) =
  ctrl(user,pass,retr,quit,ok,err).
   (ok.FTPUserSendingUSER(user,pass,retr,quit,ok,err) +
    err.FTPUser(connect) )
FTPUserSendingUSER(user,pass,retr,quit,ok,err) =
  τ.u̅s̅e̅r̅.FTPUserWaitingOk1(user,pass,retr,quit,ok,err) + τ.q̅u̅i̅t̅.0
FTPUserWaitingOk1(user,pass,retr,quit,ok,err) =
  ok.FTPUserSendingPASS(user,pass,retr,quit,ok,err)
FTPUserSendingPASS(user,pass,retr,quit,ok,err) =
  τ.p̅a̅s̅s̅.FTPUserWaitingOk2(user,pass,retr,quit,ok,err) + τ.q̅u̅i̅t̅.0
FTPUserWaitingOk2(user,pass,retr,quit,ok,err) =
  ok.FTPUserSendingCMD(retr,quit,ok,err) +
  err.FTPUserSendingUSER(user,pass,retr,quit,ok,err)
FTPUserSendingCMD(retr,quit,ok,err) =
  τ.r̅e̅t̅r̅.FTPUserWaitingOk3(retr,quit,ok,err) + τ.q̅u̅i̅t̅.0
FTPUserWaitingOk3(retr,quit,ok,err) =
  ok(data).( FTPUserDTP(data) | FTPUserSendingCMD(retr,quit,ok,err) )
FTPUserDTP(data) = data(datum,eof).FTPUserReceivingData(datum,eof)
FTPUserDTPReceivingData(datum,eof) =
  datum.FTPUserDTPReceivingData(datum,eof) + eof.0
```

Figure 4: FTPUser role

Theorem 5.5 *Let a system be formed by the parallel composition of a set of components* $\{Comp_i\}_i$*, with ports* $\mathcal{P}$*, and a set of connectors* $\{Conn_j\}_j$*, correctly specified by a set* $\mathcal{R}$ *of roles. Let* v *be the free names in* $\{Conn_j\}_j$*. Let* φ *be an architecture from* $\mathcal{P}$ *to* $\mathcal{R}$*. Then we have that* $(v)(\prod_j Conn_j \mid \prod_{P\in\mathcal{P}} P)$ *is deadlock-free.*

Proof. We can partition $(v)(\prod_j Conn_j \mid \prod_{P\in\mathcal{P}} P)$ into several $(v_j)(Conn_j|P_{j_1}|...|P_{j_{k_j}})$, each of them deadlock-free as a result of Theorem 5.3. Provided all free names in these processes are distinct, there is no possible interaction among them and therefore their parallel composition is also deadlock-free. □

6 Case Study: An FTP Server Connector

We will use an example to show the characteristics of our proposal. An FTP server communicates user processes with a remote file system, allowing the transmission of data files in both directions. Therefore it may be considered as a connector, though probably a composite one. The FTP server connector considered follows the FTP protocol as described in RFC 959 [10]. The protocol is simplified in order to avoid unnecessary complexity, but trying to maintain its main characteristics.

In order to find out if a certain user process is compatible with an FTP Server connector we will use ports and roles as abstractions of components and connectors and check their compatibility using the relation defined above. These ports and roles will pay attention

FTPUser1(connect) = (ctrl,user,pass,retr,quit,ok,err)
  $\overline{\texttt{connect}}$ ctrl.ctrl(user,pass,retr,quit,ok,err).ok.
  $\overline{\texttt{user}}$.ok.$\overline{\texttt{pass}}$.ok.$\overline{\texttt{retr}}$.ok(data).(FTPUserDTP(data) | $\overline{\texttt{quit}}$.$\mathbf{0}$)

Figure 5: An FTPUser port, not compatible

FTPUser2(connect) = (ctrl,user,pass,retr,quit,ok,err)
  $\overline{\texttt{connect}}$ ctrl.ctrl(user,pass,retr,quit,ok,err).
   (err.$\mathbf{0}$+ ok.$\overline{\texttt{user}}$.ok.$\overline{\texttt{pass}}$.
    (err.$\overline{\texttt{quit}}$.$\mathbf{0}$+ ok.$\overline{\texttt{retr}}$.ok(data).(FTPUserDTP(data) | $\overline{\texttt{quit}}$.$\mathbf{0}$)))

Figure 6: Another FTPUser port, compatible

to the significant aspects of their interaction only and will be much simpler and easier to analyze.

The FTP Server connector has two roles: one attached to the file system and the other attached to the user. The role `FTPUser` (Figure 4) describes the behavior required to processes willing to connect to a remote file system via the FTP Server. First of all, the user process must connect to a remote server sending a `connect` request to the remote host. If the connection is accepted, the protocol enters in an authentication phase, where the user process must produce a correct user name and password. These are validated and, provided they are correct, the user process is ready to send and receive data files to and from the remote host. At any point the user process may send a `quit` command and disconnect. When a `retr` command is received, a new process, the `FTPServerDTP` (data transfer process) is created. This process will send the requested file to the user as a sequence of data items (`datum`) ending with an end of file (`eof`) message.

As we can see, the user process may decide whether sending an FTP command or quitting. In contrast, when retrieving a file the `FTPServerDTP` process holds control of the communication sequence, deciding its termination by sending an eof message. Now we can check the role against a certain user port `FTPUser1` (Figure 5) and we will find out that they are not compatible. In fact, the port doesn't take into account the possibility of rejection to the `connect` command, nor considers a failure in the authentication phase (both against the third condition of the relation of compatibility). A perfectly compatible port, `FTPUser2` is specified in Figure 6.

7 Conclusions

The importance of Software Architecture is becoming more evident as software systems become more complex. However there is a lack of methods and tools specifically developed for the specification analysis and validation of the structure of software systems. Usually software architectures are described in an informal and ambiguous manner, leading to problems in the development process. In this paper we tried to show how the application of formal methods can be a solution to some of these problems. In particular, we have defined a new relation in the context of π-calculus, a very well-known process algebra. This relation models compatibility among processes represented as collections of ports and roles, and can be easily automated. The use of ports and roles permits local analysis of compatibility, thus reducing the complexity of the analysis. Mobility is easily specified in π-calculus, which makes our proposal specially interesting for dynamic systems, and therefore for distributed software systems.

References

[1] G. Abowd, R. Allen and D. Garlan. "Using Style to Understand Descriptions of Software Architecture". *Proc. SIGSOFT'93: Foundations of Software Engineering*, December 1993.

[2] R. Allen and D. Garlan. "A Formal Approach to Software Architectures". *Proc. IFIP Congress '92*, September 1992.

[3] R. Allen and D. Garlan. "Formal Connectors". *Technical Report* CMU-CS-94-115. Carnegie-Mellon, March 1994. Available at: <ftp://reports.adm.cs.cmu.edu>

[4] R. Allen and D. Garlan. "Formalizing architectural connection". *Proc. of the Sixteenth International Conference on Software Engineering*, Sorrento (Italy), May 1994, pp. 71–80.

[5] D. Garlan and D. E. Perry. "Introduction to the Special Issue on Software Architecture". *IEEE Transactions on Software Engineering, vol. 21, no. 4*, April 1995, pp. 269–274.

[6] P. Inverardi and A. L. Wolf. "Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model". *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[7] D. C. Luckham et al. "Specification and Analysis of System Architecture using Rapide". *IEEE Transactions on Software Engineering*, vol. 21, no. 4, April 1995, pp. 336–355.

[8] J. Magee et al. "Specifying Distributed Software Architectures". *Proc. of the Fifth European Software Engineering Conference ESEC'95*, Barcelona, September 1995.

[9] R. Milner, J. Parrow and D. Walker. "A Calculus of Mobile Processes, Parts I and II". *Journal of Information and Computation*, vol. 100, 1992, pp. 1–77.

[10] J. Postel and J. Reynolds. *File Transfer Protocol (FTP).* RFC 959. Information Sciencies Institute. University of Southern California, October 1985.

[11] D. Sangiorgi. "A theory of bisimulation for the π-calculus". *Technical Report* ECS-LFCS-93-270, University of Edinburgh, June 1993.

[12] M. Shaw and D. Garlan. "Characteristics of Higher-level Languages for Software Architecture". *Technical Report* CMU-CS-94-210. Carnegie-Mellon University, December 94.

[13] M. Shaw and D. Garlan. "Formulations and Formalisms in Software Architecture". J. van Leeuwen (Ed.) *Computer Science Today. LNCS 1000.* Springer Verlag, 1995, pp. 307–323.

[14] M. Shaw et al. "Abstractions for Software Architecture and Tools to Support Them". *IEEE Transactions on Software Engineering*, vol. 21, no. 4, April 1995, pp. 314–335.

[15] B. Victor. "A Verification Tool for the Polyadic π-calculus". Licenciate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994.