# On the Formalization of Architectural Types with Process Algebras [*]

Marco Bernardo
Università di Torino
Dip. di Informatica
Corso Svizzera 185
10149 Torino, Italy
bernardo@di.unito.it

Paolo Ciancarini
Università di Bologna
Dip. di Scienze dell'Inf.
Mura Anteo Zamboni 7
40127 Bologna, Italy
cianca@cs.unibo.it

Lorenzo Donatiello
Università di Bologna
Dip. di Scienze dell'Inf.
Mura Anteo Zamboni 7
40127 Bologna, Italy
donat@cs.unibo.it

## ABSTRACT

Architectural styles play an important role in software engineering as they convey codified principles and experience which help the construction of software systems with high levels of efficiency and confidence. We address the problem of formalizing and analyzing architectural styles in an operational setting by introducing the intermediate abstraction of architectural type. We develop the concept of architectural type in a process algebraic framework because of its modeling adequacy and the availability of means, such as Milner's weak bisimulation equivalence, which allow us to reason compositionally and efficiently about the well formedness of architectural types.

## 1. INTRODUCTION

Software architecture [11, 12] has emerged in the last decade as a discipline within software engineering. An important goal of this discipline is the creation of an established and shared understanding of the common forms of software design. Starting from the user requirements, the designer should be able to identify a suitable organizational style, in order to capitalize on codified principles and experience to specify, analyze, plan, and monitor the construction of a software system with high levels of efficiency and confidence. Many of these organizational patterns, usually referred to as *architectural styles*, have been developed over the years as designers recognized the value of specific organizational principles and structures for certain classes of software. Among them we have client-server systems, pipe-filter organizations, layered architectures, and so on.

An architectural style defines a family of systems having a common vocabulary of components and connectors as well as a common topology and set of contraints on the interactions among components and connectors. Given an architectural style, the set of components and connectors and their internal behavior can vary from architectural instance to architectural instance, but the structure of the overall interconnection of components and connectors and their internal behavior w.r.t. interactions is fixed.

Since an architectural style encompasses an entire family of software systems, it is desirable to formalize the concept of architectural style both to have a precise definition of the system family and to study (hopefully in an automated way) the architectural properties common to all the systems of the family. In [1] a formal framework has been provided for precisely defining architectural styles and analyzing within and between different architectural styles. This is accomplished by means of a small set of mappings from the syntactic domain of architectural descriptions to the semantic domain of architectural meaning, following the standard denotational approach developed for programming languages.

The purpose of this paper is to make a step towards the formalization of architectural styles in an operational framework suited to already existing architectural description languages (ADLs for short), in such a way that it is possible to investigate the properties which are common to all the instances of an architectural style. This is not an easy task in general, as there are at least two degrees of freedom: variability of the set of components and connectors and variability of their internal behavior. Since the internal behavior of components and connectors w.r.t. their interactions is fixed, we recognize that the above mentioned task can be made manageable if we concentrate on a restriction of the concept of architectural style – *architectural type* – which is obtained by keeping fixed the set of components and connectors and we consider those properties common to all the instances of an architectural type – *interactional properties* – which make assertions about component and connector interactions only. We show that it is possible to formalize the concept of architectural type and investigate interactional properties by means of process algebras.

The reason for the adoption of process algebras is twofold. On the one hand, they have been extensively studied in the past twenty years (see, e.g., [9, 7, 10]) and several tool supported (see, e.g., [6]) techniques have been developed for

[*]This research has been funded by Progetto MURST Cofinanziato SALADIN: "Architetture Software e Linguaggi per Coordinare Componenti Distribuite e Mobili" and by a grant from Microsoft Research Europe.

the analysis of concurrent and distributed systems described with process algebras. On the other hand, their abstraction mechanisms, their process oriented nature, and their compositionality make them well suited to be employed at the architectural level of design. This is witnessed by the fact that the ADL WRIGHT [3] relies on the process algebra CSP [7] to specify component and connector behaviors and to define the semantics of its architectural descriptions. Likewise, the ADL DARWIN [8] exploits the process algebra known as $\pi$-calculus [10] to define the semantics of its dynamic architectural interconnections.

The formalization of the concept of architectural type is carried out in this paper by means of a process algebra based ADL called PADL, which is inspired by WRIGHT and DARWIN and allows several architectures (those of the same type) to be defined at once. Each architectural type is defined as a function of its component and connector types, its topology, and its interactions. A component/connector type is in turn defined as a function of its behavior, specified as a family of process algebra terms, and its interactions, specified as a set of process algebra actions. The architectural topology consists of a set of component/connector instances related by a set of attachments among their interactions. Finally, architectural interactions are interactions of component instances which act as interactions for the whole architectural type, thus supporting hierarchical architectural modeling. The instances of a given architectural type are generated by letting the behavior of component and connector types vary. In other words, the component/connector types specified in an architectural type are viewed as being formal, so one can call for an architectural type and pass to it actual component/connector types.

PADL is provided with two methods for verifying architectural compatibility and architectural conformity, respectively. The purpose of the former check is to ensure that an architectural type is well connected, in the sense that every pair composed of a component instance and a connector instance attached to each other interact in a proper way. This is formalized by requiring that the behavior of the two instances, when projected on the interactions involved in the related attachments, is the same. The latter check, instead, aims at guaranteeing that the actual parameters are consistent with the formal ones in case of architectural type invocation. This is formalized by requiring that the actual parameters do not alter the semantics of the architectural type w.r.t. component and connector interactions, i.e. the behavior of the architectural type when projected on such interactions. Technically, both checks are carried out by means of the weak bisimulation equivalence [9] because of its ability to reason about the behavior of process algebra terms when projected on certain actions, i.e. when abstracting from unimportant actions.

A nice property of the weak bisimulation equivalence is that it is a congruence, i.e. it is substitutive, w.r.t. the so called static operators of process algebras. For instance, in the case of the parallel composition operator $E_1 \parallel_S E_2$, where $S$ is the set of synchronization actions, if term $E_1'$ is weakly bisimulation equivalent to term $E_1$, then $E_1' \parallel_S E_2$ is weakly bisimulation equivalent to the original term $E_1 \parallel_S E_2$. The congruence property of the weak bisimulation equivalence brings two advantages in our framework. First, if an architectural type is well connected, then so is each of its architectural instances obtained by passing consistent parameters. Second, the conformity of an architecture to a given architectural type can be conducted compositionally, hence efficiently. In fact, verifying architectural conformity requires in principle to build the (projected) state spaces underlying the architecture and the architectural type and checking them for weak bisimulation equivalence. Since such state spaces represent an interleaving view of the parallel composition of component and connector instances, their size grows exponentially with the number of component and connector instances, hence their construction may not be possible. Thanks to the congruence property of the weak bisimulation equivalence w.r.t. the parallel composition operator, the complexity of the architectural conformity checking can be kept linear with the number of component and connector types, as it is sufficient to build the (projected) state spaces underlying the individual component and connector types and check state spaces of pairs of corresponding component/connector types for weak bisimulation equivalence. In other words, the compositional process above permits to efficiently establish whether the architecture and the architectural type possess the same interactional properties.

We observe that our approach to formalizing architectural styles via the intermediate abstraction of architectural type is complementary to a recent extension of WRIGHT supporting architectural styles (see, e.g., [4]). In that approach, the description of an architectural style only comprises component/connector types with a fixed internal behavior as well as topological constraints, whereas component/connector instances and the related attachments are separately specified in configurations of the style. While that approach allows the set of component/connector instances and the related attachments to vary from configuration to configuration in a controlled way, in our approach such a set is fixed and specified in the description of the architectural type together with component/connector types whose behavior can vary in a controlled way, so that the interactional properties of an architectural type can be analyzed.

This paper is organized as follows. In Sect. 2 we present some background on process algebras and the weak bisimulation equivalence which is necessary to understand the rest of the paper. In Sect. 3 we present the syntax and the semantics for PADL and we make some comparison with WRIGHT and DARWIN. In Sect. 4 we introduce a weak bisimulation equivalence based technique for checking architectural compatibility. In Sect. 5 we explain architectural parameter passing and we show a weak bisimulation equivalence based technique for efficiently checking architectural conformity. Finally, in Sect. 6 we report some concluding remarks.

## 2. PROCESS ALGEBRAS

Process algebras [9, 7] 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 its actions, which represent activities carried out by the systems being modeled, and its operators (among which a parallel composition operator), which are used to compose algebraic descriptions. In this section we introduce an example pro-

$$a.E \xrightarrow{a} E$$

$$\frac{E \xrightarrow{a} E'}{E/L \xrightarrow{a} E'/L} \text{ if } a \notin L \qquad \frac{E \xrightarrow{a} E'}{E/L \xrightarrow{\tau} E'/L} \text{ if } a \in L$$

$$\frac{E \xrightarrow{a} E'}{E[\varphi] \xrightarrow{\varphi(a)} E'[\varphi]}$$

$$\frac{E_1 \xrightarrow{a} E'}{E_1 + E_2 \xrightarrow{a} E'} \qquad \frac{E_2 \xrightarrow{a} E'}{E_1 + E_2 \xrightarrow{a} E'}$$

$$\frac{E_1 \xrightarrow{a} E'_1}{E_1 \parallel_S E_2 \xrightarrow{a} E'_1 \parallel_S E_2} \text{ if } a \notin S \qquad \frac{E_2 \xrightarrow{a} E'_2}{E_1 \parallel_S E_2 \xrightarrow{a} E_1 \parallel_S E'_2} \text{ if } a \notin S$$

$$\frac{E_1 \xrightarrow{a} E'_1 \quad E_2 \xrightarrow{a} E'_2}{E_1 \parallel_S E_2 \xrightarrow{a} E'_1 \parallel_S E'_2} \text{ if } a \in S$$

$$\frac{E \xrightarrow{a} E'}{A \xrightarrow{a} E'} \text{ if } A \triangleq E$$

**Table 1: Operational semantics for PA**

cess algebra called PA on which we shall construct an ADL for architectural types.

The set of process terms of PA is generated by the following syntax

$$E ::= \underline{0} \mid a.E \mid E/L \mid E[\varphi] \mid E + E \mid E \parallel_S E \mid A$$

where $a$ belongs to a set *Act* of actions including a distinguished action $\tau$ for unobservable activities, $L, S \subseteq Act - \{\tau\}$, $\varphi$ belongs to a set *ARFun* of action relabeling functions preserving obervability (i.e., $\varphi^{-1}(\tau) = \{\tau\}$), and $A$ belongs to a set *Const* of constants each possessing a (possibly recursive) defining equation of the form $A \triangleq E$.

In the syntax above, the null term "$\underline{0}$" is the term that cannot execute any action. The prefix operator "$a.\_$" denotes the sequential composition of an action and a term: term $a.E$ can execute action $a$ and then behaves as term $E$. The hiding operator "$\_/L$" hides actions: term $E/L$ behaves as term $E$ except that each executed action $a$ is turned into $\tau$ whenever $a \in L$. The relabeling operator "$\_[\varphi]$" changes actions: term $E[\varphi]$ behaves as term $E$ except that each executed action $a$ is turned into $\varphi(a)$. The alternative composition operator "$\_ + \_$" expresses a nondeterministic choice between two terms: term $E_1 + E_2$ behaves as either term $E_1$ or term $E_2$ depending on whether an action of $E_1$ or an action of $E_2$ is executed. The parallel composition operator "$\_ \parallel_S \_$" expresses the concurrent execution of two terms according to the following synchronization discipline: two actions can synchronize if and only if they are equal and observable. Term $E_1 \parallel_S E_2$ asynchronously executes actions of $E_1$ or $E_2$ not belonging to $S$ and synchronously executes actions of $E_1$ and $E_2$ belonging to $S$ if the requirement above is met.

The prefix operator and the alternative composition operator are called dynamic operators, whereas the hiding operator, the relabeling operator, and the parallel composition operator are called static operators. We say that a term is sequential if it is composed of dynamic operators only.

The semantics for PA can be defined in the standard operational style by means of a set of axioms and inference rules which formalize the meaning of each operator and produce as semantic model a state transition graph where states are in correspondence with process terms and transitions are labeled with actions. The semantics for PA is shown in Table 1. As an example, the first rule for the hiding operator means that, if the current state is $E/L$ and $E$ is capable of performing an action $a \notin L$ thus evolving into $E'$, then $E/L$ is capable of performing the same action $a$ thus evolving into $E'/L$. The other rules must be read in the same way. In particular, note that in the third rule for the parallel composition operator the action resulting from the synchronization of two $a$ actions is still $a$. This means that multiway synchronizations are allowed.

Due to their algebraic nature, process description languages like PA naturally lend themselves to the definition of equivalences and preorders. In particular, in the concurrency theory literature several notions of equivalence can be found which relate terms according to a certain interpretation of their behavior. Here we recall the weak bisimulation equivalence of [9], which captures the ability of two terms to simulate each other behavior up to $\tau$ actions.

Let us generalize the transition relation $\longrightarrow$ labeled with actions to the transition relation $\Longrightarrow$ labeled with sequences of actions: $\xRightarrow{a_1 \dots a_n} \equiv \xrightarrow{a_1} \dots \xrightarrow{a_n}$, with $\varepsilon$ denoting the empty string and $\xRightarrow{\varepsilon}$ being the identity relation over process terms. Moreover, if $\sigma$ is a sequence over $Act$, let $\hat{\sigma}$ be the sequence over $Act - \{\tau\}$ obtained from $\sigma$ by removing all the occurring $\tau$ actions. Finally, we use $\overset{\sigma}{\models\!\!\Longrightarrow} \equiv \xRightarrow{\tau^m} \xRightarrow{\sigma} \xRightarrow{\tau^n}$, with $m, n \in \mathbb{N}$, to indicate the execution of an action sequence $\sigma$ possibly preceded and followed by the execution of arbitrarily many invisible actions. A binary relation $\mathcal{B}$ over process terms is a weak bisimulation if and only if, whenever $(E_1, E_2) \in \mathcal{B}$, then for all $a \in Act$:

- whenever $E_1 \xrightarrow{a} E'_1$, then $E_2 \overset{\hat{a}}{\models\!\!\Longrightarrow} E'_2$ and $(E'_1, E'_2) \in \mathcal{B}$ for some $E'_2$;
- whenever $E_2 \xrightarrow{a} E'_2$, then $E_1 \overset{\hat{a}}{\models\!\!\Longrightarrow} E'_1$ and $(E'_1, E'_2) \in \mathcal{B}$ for some $E'_1$.

Note that $\hat{a} = a$ for $a \neq \tau$, $\hat{a} = \varepsilon$ for $a = \tau$. The union of all the weak bisimulations, denoted by $\approx$, is called the weak bisimulation equivalence.

$\approx$ enjoys several algebraic properties. For instance, the properties w.r.t. the dynamic operators are listed below:

$$\begin{array}{rcl}
(E_1 + E_2) + E_3 & \approx & E_1 + (E_2 + E_3) \\
E_1 + E_2 & \approx & E_2 + E_1 \\
E + \underline{0} & \approx & E \\
E + E & \approx & E \\
a.\tau.E & \approx & a.E \\
E + \tau.E & \approx & \tau.E \\
a.(E_1 + \tau.E_2) + a.E_2 & \approx & a.(E_1 + \tau.E_2) \\
\tau.E & \approx & E
\end{array}$$

The last four properties are those which show the ability of $\approx$ to abstract from $\tau$ actions and make (the use of the hiding operator and) $\approx$ suited to reason on projections of the behavior of architectural components and connectors. Moreover, $\approx$ is a congruence, i.e. it is substitutive, w.r.t. the static operators: if $E_1 \approx E_2$ then

$$\begin{array}{rcl}
E_1/L & \approx & E_2/L \\
E_1[\varphi] & \approx & E_2[\varphi] \\
E_1 \parallel_S E & \approx & E_2 \parallel_S E
\end{array}$$

This property will be exploited to devise an efficient procedure for checking architectural conformity.

## 3. A PROCESS ALGEBRA BASED ADL

In this section we present the syntax and the semantics for PADL, an ADL based on PA introduced in the previous section, and we make some comparison with the ADLs Wright and Darwin by which it is inspired. For the sake of simplicity, the presentation will be mainly based on a running example concerning a pipe-filter architecture. However, the general case will be treated as well.

### 3.1 Syntax

A description in PADL is an architectural type. Each architectural type is defined as a function of its component and connector types, its topology, and its interactions. A component/connector type is in turn defined as a function of its behavior, specified either as a family of PA sequential terms or through an invocation of a previously specified architectural type, and its interactions, specified as a set of PA actions. The architectural topology consists of a fixed set of component/connector instances related by a fixed set of attachments among their interactions. Finally, architectural interactions are interactions of component instances which act as interactions for the whole architectural type, thus supporting hierarchical architectural modeling.

From the PA perspective, creating PADL can be viewed as an attempt to force the designer to model systems in a more controlled way, which in particular elucidates the basic architectural concepts of component and connector, thus hopefully enhancing the usability of PA. Since this syntactic sugar alone is not enough to create a useful ADL out of PA, PADL is equipped with suitable techniques to verify the well formedness of architectural descriptions, such as the architectural compatibility and conformity checks we shall explain in Sect. 4 and 5, respectively. In order to enforce the controlled way of modeling mentioned above, the distinction between dynamic and static operators of PA is exploited. More precisely, only the (easier) dynamic operators can be employed when defining the behavior of components and connectors that are atomic, i.e. that are not instances of some architectural type. Thus, the behavior of atomic components and connectors is modeled via PA sequential terms. Instead, the (more difficult) static operators are employed to define the semantics and the related analysis techniques, so they are transparent to the architect. As we shall see, the parallel composition operator is used in the definition of the semantics of an architectural type as it realizes the structure of a system according to the specified attachments between component and connector interactions. The hiding operator is instead used in the architectural compatibility checking and in the architectural conformity checking to derive the projection of the behavior of component and connector instances w.r.t. their interactions. Finally, the relabeling operator is used in conjunction with the parallel composition operator and the hiding operator only for technical reasons.

An architectural description in PADL is as shown in Table 2. We now illustrate the meaning of each of the parts of the architectural description above by means of an example concerning a pipe-filter architectural type. It is composed of three identical filters and a pipe which forwards items processed by the upstream filter to one of the two downstream filters for further processing. Each filter acts as a service center with a two position buffer. For each item processed by the upstream filter, the pipe forwards it to one of the two downstream filters according to the availability of free positions in their buffers. If both have free positions, the choice is resolved nondeterministically.

The first part of a PADL description defines the name of the architectural type and its formal parameters, i.e. component types, connector types, and architectural interactions:

| | |
|---|---|
| **archi_type** | ⟨name and formal parameters⟩ |
| **archi_cc_types** | ⟨architectural component/connector types⟩ |
| **archi_topology** | ⟨architectural topology: component/connector instances and attachments⟩ |
| **archi_interactions** | ⟨architectural interactions⟩ |
| **end** | |

**Table 2: PADL textual notation**

$$\textbf{archi\_type} \quad \mathit{PipeFilter}(\mathit{FilterT};\ \mathit{PipeT};\ \mathit{PipeFilterI})$$

The specification above indicates that the architectural type named *PipeFilter* relies on a single component type $\mathit{FilterT}$, a single connector type $\mathit{PipeT}$, and a set $\mathit{PipeFilterI}$ of architectural interactions.

The second part of the description defines the component and connector types which have been specified to be formal parameters of the architectural type in the first part. Each component/connector type is defined through a family of PA sequential terms (or an invocation of a previously specified architectural type) expressing its behavior and a set of PA actions expressing its interactions:

$$\begin{array}{l}
\textbf{archi\_cc\_types} \\
\quad \textbf{component\_type}\ \mathit{FilterT}(\mathit{Filter};\ \mathit{FilterI}) \\
\quad\quad \textbf{behavior} \\
\quad\quad\quad \mathit{Filter} \triangleq \mathit{accept\_item}.\mathit{Filter}' \\
\quad\quad\quad \mathit{Filter}' \triangleq \mathit{accept\_item}.\mathit{Filter}'' + \mathit{serve\_item}.\mathit{Filter} \\
\quad\quad\quad \mathit{Filter}'' \triangleq \mathit{serve\_item}.\mathit{Filter}' \\
\quad\quad \textbf{interactions}\ \mathit{FilterI} = \mathit{accept\_item}, \mathit{serve\_item} \\
\quad \textbf{connector\_type}\ \mathit{PipeT}(\mathit{Pipe},\ \mathit{PipeI}) \\
\quad\quad \textbf{behavior} \\
\quad\quad\quad \mathit{Pipe} \triangleq \mathit{accept\_item}.(\mathit{forward\_item}_1.\mathit{Pipe} + \mathit{forward\_item}_2.\mathit{Pipe}) \\
\quad\quad \textbf{interactions}\ \mathit{PipeI} = \mathit{accept\_item}, \mathit{forward\_item}_1, \mathit{forward\_item}_2
\end{array}$$

The specification above indicates that the component type $\mathit{FilterT}$ has the behavior described by term *Filter*, which represents a service center of capacity two, and the interactions described by action set *FilterI*. Initially, the filter can only accept an item from the outside, thus evolving into $\mathit{Filter}'$. If an item is in the buffer ($\mathit{Filter}'$), then either another item from the outside is accepted or the item in the buffer is served. If two items are in the buffer ($\mathit{Filter}''$), no more items can be accepted; the filter can only serve one of the waiting items. Similarly, the connector type $\mathit{PipeT}$ has the behavior described by term *Pipe* and the interactions described by action set *PipeI*. The pipe repeatedly accepts an item and forwards it along one of two different routes.

The third part of the description defines the (fixed) topology of the architectural type in terms of instances of the previously introduced component and connector types and related attachments among their interactions:

$$\begin{array}{ll}
\textbf{archi\_topology} & F_0, F_1, F_2 : \mathit{FilterT} \\
& P : \mathit{PipeT} \\
& F_0.\mathit{serve\_item}\ \textbf{to}\ P.\mathit{accept\_item} \\
& F_1.\mathit{accept\_item}\ \textbf{to}\ P.\mathit{forward\_item}_1 \\
& F_2.\mathit{accept\_item}\ \textbf{to}\ P.\mathit{forward\_item}_2
\end{array}$$

For each attachment, the first element is a component instance interaction while the second element is a connector instance interaction. The attachments above specify that an instance $P$ of $\mathit{PipeT}$ is connected to an upstream instance $F_0$ of $\mathit{FilterT}$ and two downstream instances $F_1, F_2$ of $\mathit{FilterT}$.

Finally, the fourth part defines the architectural interactions. They are interactions of the previously defined component instances which are used when defining the behavior of a component/connector type of another architectural type by invoking the architectural type being specified, i.e. when plugging the architectural type at hand in the context of a larger system:

$$\textbf{archi\_interactions} \quad \mathit{PipeFilterI} = F_0.\mathit{accept\_item}, F_1.\mathit{serve\_item}, F_2.\mathit{serve\_item}$$

This part supports hierarchical modeling. For instance, the pipe-filter architecture may represent the structure of the server in a client-server architecture.

## 3.2 Semantics

Let us now define the semantics of the PADL specification considered in the previous section. We define the semantics of a component/connector type to be the term expressing its behavior:

$$\begin{array}{rcl}
[\![\mathit{FilterT}]\!] & = & \mathit{Filter} \\
[\![\mathit{PipeT}]\!] & = & \mathit{Pipe}
\end{array}$$

Likewise, we define the semantics of a component/connector instance to be the semantics of the related type:

$$\begin{array}{rcl}
[\![F_0]\!] & = & [\![\mathit{FilterT}]\!] \\
[\![F_1]\!] & = & [\![\mathit{FilterT}]\!] \\
[\![F_2]\!] & = & [\![\mathit{FilterT}]\!] \\
[\![P]\!] & = & [\![\mathit{PipeT}]\!]
\end{array}$$

The semantics of the whole architectural type is then obtained by composing in parallel the semantics of the component/connector instances according to the specified attachments:

$$\begin{array}{rcl}
[\![\mathit{PipeFilter}]\!] & = & [\![F_0]\!][\mathit{serve\_item} \mapsto a] \,\|_{\{a\}} \\
& & [\![P]\!][\mathit{accept\_item} \mapsto a] \\
& & [\mathit{forward\_item}_1 \mapsto a_1, \\
& & \mathit{forward\_item}_2 \mapsto a_2] \,\|_{\{a_1, a_2\}} \\
& & ([\![F_1]\!][\mathit{accept\_item} \mapsto a_1] \,\|_{\emptyset} \\
& & [\![F_2]\!][\mathit{accept\_item} \mapsto a_2])
\end{array}$$

The use of the relabeling operator is necessary to make instances interact. As an example, $F_0$ and $P$ must interact via *serve_item* and *accept_item* which are different from each other. Since the parallel composition operator allows actions to synchronize only if they are equal, in $\llbracket \mathit{PipeFilter} \rrbracket$ above each *serve_item* executed by $\llbracket F_0 \rrbracket$ and each *accept_item* executed by $\llbracket P \rrbracket$ is relabeled to the same action $a$ (occurring neither in $\llbracket F_0 \rrbracket$ nor in $\llbracket P \rrbracket$ to avoid interferences), then a synchronization on $a$ is forced between the relabeled versions of $\llbracket F_0 \rrbracket$ and $\llbracket P \rrbracket$ by means of operator $\|_{\{a\}}$.

In general, the formal semantics for PADL is given by translation into PA by essentially exploiting the parallel composition operator. Given a PADL specification in which $m$ component instances $C_1, \dots, C_m$ and $n$ connector instances $K_1, \dots, K_n$ are declared, and defined their individual semantics as the PA terms expressing the behavior of the corresponding component/connector types, for the sake of simplicity we consider two different cases depending on the attachments $I_1, \dots, I_p$ declared in the specification.

If every interaction is involved in at most one attachment, then only binary synchronizations are possible, hence the component instances are independent of each other and the connector instances are independent of each other. Let $S = \{a_1, \dots, a_p\}$ be a set of as many fresh actions as there are attachments, where fresh means that they do not occur in $\llbracket C_1 \rrbracket, \dots, \llbracket C_m \rrbracket, \llbracket K_1 \rrbracket, \dots, \llbracket K_n \rrbracket$. Moreover, for each attachment $I_i$, say $C_j.b$ **to** $K_k.c$, let us apply relabeling $[b \mapsto a_i]$ to $\llbracket C_j \rrbracket$ and relabeling $[c \mapsto a_i]$ to $\llbracket K_k \rrbracket$. For each term $\llbracket C_i \rrbracket$ ($\llbracket K_i \rrbracket$), let us denote by $\llbracket C_i \rrbracket'$ ($\llbracket K_i \rrbracket'$) the relabeled version of $\llbracket C_i \rrbracket$ ($\llbracket K_i \rrbracket$) according to all the attachments involving the interactions of $C_i$ ($K_i$). Then the semantics of the specification is given by

$$(\llbracket C_1 \rrbracket' \|_{\emptyset} \dots \|_{\emptyset} \llbracket C_m \rrbracket') \|_S (\llbracket K_1 \rrbracket' \|_{\emptyset} \dots \|_{\emptyset} \llbracket K_n \rrbracket')$$

Following this scheme, the semantics of *PipeFilter* can be rephrased as

$$(\llbracket F_0 \rrbracket' \|_{\emptyset} \llbracket F_1 \rrbracket' \|_{\emptyset} \llbracket F_2 \rrbracket') \|_{\{a, a_1, a_2\}} \llbracket P \rrbracket'$$

where

$$\begin{array}{rcl} \llbracket F_0 \rrbracket' & = & \llbracket F_0 \rrbracket[\mathit{serve\_item} \mapsto a] \\ \llbracket F_1 \rrbracket' & = & \llbracket F_1 \rrbracket[\mathit{accept\_item} \mapsto a_1] \\ \llbracket F_2 \rrbracket' & = & \llbracket F_2 \rrbracket[\mathit{accept\_item} \mapsto a_2] \\ \llbracket P \rrbracket' & = & \llbracket P \rrbracket[\mathit{accept\_item} \mapsto a, \\ & & \quad \mathit{forward\_item}_1 \mapsto a_1, \\ & & \quad \mathit{forward\_item}_2 \mapsto a_2] \end{array}$$

If there are interactions involved in more than one attachment, instead, also multiway synchronizations are possible, which means that there are component/connector instances that are not independent of each other. To characterize such multiway synchronizations, we say that two interactions $a_1, a_2$ are connected if either there is an attachment between them, or there is an interaction $a$ such that $a_1$ is connected to $a$ and there is an attachment between $a$ and $a_2$. Moreover, we say that a subset of attachments is connected if the involved interactions are pairwise connected via attachments of the subset only and the subset is maximal. Let $S = \{a_1, \dots, a_q\}$, $q \leq p$, be a set of as many fresh actions as there are connected subsets of attachments, let $S(C_i, C_j)$ ($S(K_i, K_j)$) be the subset of actions of $S$ corresponding to all and only the connected subsets of attachments involving $C_i$ and $C_j$ ($K_i$ and $K_j$), and let $\llbracket C_i \rrbracket'$ ($\llbracket K_i \rrbracket'$) have the same meaning as in the previous case. Then, recalling that the parallel composition operator is left associative, the semantics of the specification is given by

$$\begin{array}{c} (\llbracket C_1 \rrbracket' \|_{S(C_1,C_2)} \cdots \|_{\cup_{i=1}^{m-1} S(C_i,C_m)} \llbracket C_m \rrbracket') \|_S \\ (\llbracket K_1 \rrbracket' \|_{S(K_1,K_2)} \cdots \|_{\cup_{i=1}^{n-1} S(K_i,K_n)} \llbracket K_n \rrbracket') \end{array}$$

This case constitutes a generalization of the previous one: if every interaction is involved in at most one attachment, then all the synchronization sets above are empty (except $S$).

### 3.3 Comparison

We conclude with a comparison between PADL and the ADLs Wright and Darwin. This comparison does not take style related issues into account; these are deferred to Sect. 5.

First of all we observe that the syntax for PADL is closely related to that for Wright. In particular, we note that connectors are treated as first class entities and that the behavior of components and connectors is fully specified through process algebra terms. Moreover, like in Wright, also in PADL the semantics of an architectural description is defined by exploiting in an essential way the parallel composition operator. After all, this stems from the observation that architecture is inherently about putting parts together and the parallel composition operator is well suited for this task. Furthemore, like in Wright, also in PADL there is a technique for verifying architectural compatibility, as we shall see in Sect. 4.

However, there is a fundamental difference between PADL and Wright, which lies in the way in which interactions are specified. In Wright, the description of each component type is accompanied by the specification of its ports, which are process algebra terms expressing logical points of interaction between the component and its environment. Similarly, the description of each connector type is accompanied by the specification of its roles, which are process algebra terms expressing the expected local behavior of the components interacting through that connector. In other words, ports and roles represent projections of the related component resp. connector behavior w.r.t. certain interactions. As a consequence, Wright architectural descriptions present a certain degree of redundance, because the projected behaviors expressed by the terms representing ports and roles should (at least partly) be contained in the terms representing the overall behavior of the related components resp. connectors. Moreover, in the framework of Wright the verification of the relationship between a port/role and the related component/connector seems not to be easy. In PADL, instead, component and connector interactions are simply expressed as actions. This is sufficient from the technical viewpoint, as the parallel composition operator realizes the communication among terms through the synchronization among actions, as well as from the intuitive viewpoint, as the details of the interaction among several communicating components is already encoded in the behavior of the related connector. Furthemore, this choice leads to the absence of redundance in architectural specifications and easily allows one to (at least partly) retrieve the projected behaviors specified in Wright through ports and roles using the hiding operator. As an example, if in the architectural description of *PipeFilter* one would like to retrieve the behavior of $F_0$ w.r.t. its interaction with $P$, one should simply consider $[\![F_0]\!]/(Act - \{serve\_item, \tau\})$, i.e. the term representing the behavior of $F_0$ where every action is hidden except *serve_item*. As we shall see in the next two sections, the easy retrievability of projected behaviors, besides avoiding the architect to have to explicitly provide them, will be exploited to develop two simple techniques for checking architectural compatibility and architectural conformity, respectively, in PADL.

Another important difference between PADL and WRIGHT is that PADL supports hierarchical architectural modeling thanks to the provision of architectural interactions.

We observe that the specification of interactions through actions instead of process terms as well as the possibility of expressing architectural interactions can be found in DARWIN. However, unlike PADL, in DARWIN connectors are not treated as first class entities and the behavior of components is not fully specified.

## 4. ARCHITECTURAL COMPATIBILITY

Similarly to type checking for conventional programming languages, we present in this section a method for checking the architectural compatibility in PADL. The objective is to ensure that an architectural type is well connected, in the sense that every pair composed of a component instance and a connector instance attached to each other interact in a proper way.

Given a PADL specification, let $C$ be one of its component instances and $K$ one of its connector instances, and let $\mathcal{I}$ be the set of all the attachments involving $C$ and $K$. We say that $C$ and $K$ are compatible if their behaviors, when projected on the interactions mentioned in $\mathcal{I}$, are the same w.r.t. $\approx$. As observed in the previous section, such projections can be easily obtained through the hiding operator, hence there is no need to explicitly declare them like in WRIGHT. We then say that an architectural type is well connected, i.e. meets architectural compatibility, if all of its component and connector instances related by some attachment are compatible.

As an example, the architectural type *PipeFilter* defined in Sect. 3 meets architectural compatibility because:

$$\begin{array}{l}
[\![F_0]\!]/(Act - \{serve\_item, \tau\})[serve\_item \mapsto a] \approx \\
\quad [\![P]\!]/(Act - \{accept\_item, \tau\})[accept\_item \mapsto a] \\
[\![F_1]\!]/(Act - \{accept\_item, \tau\})[accept\_item \mapsto a_1] \approx \\
\quad [\![P]\!]/(Act - \{forward\_item_1, \tau\})[forward\_item_1 \mapsto a_1] \\
[\![F_2]\!]/(Act - \{accept\_item, \tau\})[accept\_item \mapsto a_2] \approx \\
\quad [\![P]\!]/(Act - \{forward\_item_2, \tau\})[forward\_item_2 \mapsto a_2]
\end{array}$$

where the three relationships above correspond to the three attachments declared in *PipeFilter*, respectively. Note that, analogously to the definition of the semantics of an architectural type, the use of the relabeling operator is necessary to make $\approx$ applicable. As an example, $F_0$ and $P$ must be compared when their behavior is projected onto *serve_item* and *accept_item*, respectively. Since these actions are different from each other, they are relabeled to the same action $a$ (occurring neither in $[\![F_0]\!]$ nor in $[\![P]\!]$ to avoid interferences) before applying $\approx$.

We conclude by observing that the checks above can be conducted efficiently by means of tools like [6], as they require the construction of the (projected) state spaces of individual component/connector instances instead of the state space of the whole architectural type.

## 5. ARCHITECTURAL CONFORMITY

Unlike WRIGHT and DARWIN, a description in PADL does not specify the architecture of a single software system, but the architecture of a family of systems possessing the same set of component and connector instances and related attachments. This means that the component/connector types specified in an architectural type are viewed as being formal, so one can call for an architectural type and pass to it actual component/connector types and architectural interactions in the proper order. Actual parameters should be consistent with formal ones, so that the different instances of an architectural type possess the same interactional properties and these can be investigated on the description of the architectural type itself.

As an example, the architectural type *PipeFilter* defined in Sect. 3 could be invoked as follows while specifying the behavior of a component/connector type (of another architectural type) which receives requests from the outside and fulfills them:

$$\begin{array}{l}
PipeFilter(NewFilterT; \\
\qquad\quad PipeT; \\
\qquad\quad NewPipeFilterI) \\
\textbf{where} \\
\quad \textbf{component\_type}\ NewFilterT(NewFilter; \\
\qquad\qquad\qquad\qquad\qquad\quad NewFilterI) \\
\qquad \textbf{behavior} \\
\qquad\quad NewFilter \triangleq receive\_item.NewFilter' + \\
\qquad\qquad\qquad\quad fail.repair.NewFilter \\
\qquad\quad NewFilter' \triangleq receive\_item.NewFilter'' + \\
\qquad\qquad\qquad\quad process\_item.NewFilter + \\
\qquad\qquad\qquad\quad fail.repair.NewFilter' \\
\qquad\quad NewFilter'' \triangleq process\_item.NewFilter' + \\
\qquad\qquad\qquad\quad fail.repair.NewFilter'' \\
\qquad \textbf{interactions}\ NewFilterI = receive\_item, \\
\qquad\qquad\qquad\qquad\qquad process\_item \\
\quad \textbf{archi\_interactions}\ NewPipeFilterI = receive\_request, \\
\qquad\qquad\qquad\qquad\qquad\qquad send\_outcome, \\
\qquad\qquad\qquad\qquad\qquad\qquad send\_outcome
\end{array}$$

In the invocation above, the type of pipe is unchanged, whereas a new type of filter is considered which is no longer perfect, i.e. it is possibly subject to failures and subsequent repairs. The semantics of the invocation is given by:

$$[\![PipeFilter]\!]\{NewFilterT/FilterT\}$$
$$[PipeFilterI \mapsto NewPipeFilterI]$$

where $[\![PipeFilter]\!]\{NewFilterT/FilterT\}$ is the term that is obtained from $[\![PipeFilter]\!]$ by replacing each name of *FilterT* with the corresponding name of *NewFilterT*; each action $F_0.receive\_item$, $F_1.process\_item$, and $F_2.process\_item$ executed by such a term is then relabeled to *receive_request*, *send_outcome*, and *send_outcome*, respectively. It is worth observing that $PipeFilter(FilterT; PipeT; PipeFilterI)$ and $PipeFilter(NewFilterT; PipeT; NewPipeFilterI)$ possess the same interactional properties, as their intended semantics is given by the equivalence class w.r.t. $\approx$ of the projection (up to suitable relabelings) of $[\![PipeFilter]\!]$ on its component/connector instance interactions: $[\![PipeFilter]\!]/(Act -$

$\{F_0.accept\_item, a, a_1, a_2, F_1.serve\_item, F_2.serve\_item, \tau\})$. However, their actual behaviors are different; e.g., we have that $PipeFilter(FilterT; PipeT; PipeFilterI)$ is more efficient than $PipeFilter(NewFilterT; PipeT; NewPipeFilterI)$.

In general, when invoking an architectural type, we have to make sure that its semantics, intended as the projection of its behavior on its component/connector instance interactions, is not altered. In other words, we have to make sure that we do not abandon the equivalence class w.r.t. $\approx$ of the projection of the behavior of the architectural type on its component/connector instance interactions. Thanks to the congruence property of $\approx$ w.r.t. the static operators of PA, the verification of the conformity of an architecture to a given architectural type can be conducted in a time which grows linearly with the number of component and connector types. In fact, it is not necessary to build and check for $\approx$ the (projected) state spaces underlying the architecture and the architectural type, but it is sufficient to build the (projected) state spaces underlying the individual component and connector types and check state spaces of pairs of corresponding component/connector types for $\approx$. More precisely, parameter consistency amounts to verifying (through tools like [6]) that the behavior of each actual component/connector type and the behavior of the corresponding formal component/connector type, when projected on the related interactions, are the same w.r.t. $\approx$.

As an example, the invocation above of the architectural type *PipeFilter* is consistent because:

$$[\![NewFilterT]\!]/(Act - \{receive\_item, process\_item, \tau\})\,[receive\_item \mapsto b, process\_item \mapsto c] \approx$$
$$[\![FilterT]\!]/(Act - \{accept\_item, serve\_item, \tau\})\,[accept\_item \mapsto b, serve\_item \mapsto c]$$

Note once again the use of the relabeling operator for technical reasons.

It is worth observing that if an architectural type possesses certain interactional properties (such as architectural compatibility), then they are automatically inherited by each architecture which conforms to it. In other words, the compositional checking procedure above permits to efficiently establish whether an architecture possesses the same interactional properties as a given architectural type.

We conclude by recalling from the introduction that our approach to formalizing architectural styles via the intermediate abstraction of architectural type is complementary to a recent extension of WRIGHT supporting architectural styles (see, e.g., [4]). In that approach, the description of an architectural style only comprises component/connector types as well as topological constraints, whereas component/connector instances and the related attachments are separately specified in configurations of the style. In PADL, instead, the description of an architectural type comprises both component/connector types whose behavior can vary in a controlled way and a fixed set of component/connector instances and attachments, so that the interactional properties of an architectural type can be analyzed.

## 6. CONCLUSION

In this paper we have addressed the problem of formalizing architectural styles by introducing the intermediate abstraction of architectural type: the behavior of component/connector types can vary (in a controlled way) from architectural instance to architectural instance, but the architectural topology (i.e. the set of component/connector instances and the related attachments) is fixed. This has been accomplished in a process algebraic framework because of its abstraction mechanisms, its process oriented nature, and its compositionality which make it suited to be employed at the architectural level. The resulting language PADL, inspired by both WRIGHT and DARWIN, comes equipped with a technique for verifying architectural compatibility as well as an efficient technique for verifying architectural conformity and allows a certain kind of properties common to all the instances of an architectural type (interactional properties) to be investigated.

As far as future work is concerned, first of all we would like to investigate whether our view of architectural style (controlled variability of component/connector type behavior) can be reconciled with that of a recent extension of WRIGHT (controlled variability of the set of component/connector instances and related attachments), in order to obtain a more accurate and flexible formalization of the concept of architectural style.

Moreover, we would like to investigate ways of enhancing the expressiveness of PADL in order to enlarge the class of architectural styles that can be modeled and the class of properties that can be analyzed. It is worth doing this in a process algebraic framework because in the literature there are several extensions of classical process algebras like [9, 7] dealing with mobility, security, real time, and performance. As far as mobility is concerned, we recall that DARWIN and a recent extension of WRIGHT [2] support the formalization of dynamic software architectures. In [5], instead, we have done some work on the performance evaluation of architectural types through a stochastic process algebra based ADL called ÆMPA. We are now in the process of implementing a software tool for the functional and performance analysis of well formed architectural types specified with the textual or graphical notation of ÆMPA. Such a tool will allow us to conduct some case studies to assess the adequacy of our approach. For the time being, we are using a prototype of the tool to investigate the properties of several load distribution algorithms for replicated web services in different architectural scenarios.

## 7. REFERENCES

[1] G.D. Abowd, R. Allen, D. Garlan, *"Formalizing Style to Understand Descriptions of Software Architecture"*, in ACM Trans. on Software Engineering and Methodology 4:319-364, 1995

[2] R. Allen, R. Douence, D. Garlan, *"Specifying and Analyzing Dynamic Software Architectures"*, in Proc. of the *1st Int. Conf. on Fundamental Approaches to Software Engineering (FASE '98)*, LNCS 1382:21-37, Lisbon (Portugal), 1998

[3] R. Allen, D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM Trans. on Software Engineering and Methodology 6:213-249, 1997

[4] R. Allen, D. Garlan, *"A Case Study in Architectural Modelling: The AEGIS System"*, in Proc. of the *8th Int. Workshop on Software Specification and Design (IWSSD-8)*, Paderborn (Germany), 1998

[5] M. Bernardo, P. Ciancarini, L. Donatiello, *“Performance Analysis of Software Architectures via a Process Algebraic Description Language”*, to appear in Proc. of the *2nd ACM Int. Workshop on Software and Performance (WOSP ’00)*, ACM Press, Ottawa (Canada), 2000

[6] W.R. Cleaveland, J. Parrow, B. Steffen, *“The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems”*, in ACM Trans. on Programming Languages and Systems 15:36-72, 1993

[7] C.A.R. Hoare, *“Communicating Sequential Processes”*, Prentice Hall, 1985

[8] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, *“Specifying Distributed Software Architectures”*, in Proc. of the *5th European Software Engineering Conf. (ESEC ’95)*, LNCS 989:137-153, Barcelona (Spain), 1995

[9] R. Milner, *“Communication and Concurrency”*, Prentice Hall, 1989

[10] R. Milner, J. Parrow, D. Walker, *“A Calculus of Mobile Processes”*, in Information and Computation 100:1-77, 1992

[11] D.E. Perry, A.L. Wolf, *“Foundations for the Study of Software Architecture”*, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992

[12] M. Shaw, D. Garlan, *“Software Architecture: Perspectives on an Emerging Discipline”*, Prentice Hall, 1996