On the Usability of Process Algebra: An Architectural View

Alessandro Aldini, Marco Bernardo *

Università di Urbino "Carlo Bo"
Istituto di Scienze e Tecnologie dell'Informazione
Piazza della Repubblica 13, 61029 Urbino, Italy

Abstract

Despite its strengths like compositionality and equivalence checking, process algebra is rarely adopted outside the academia. In this paper we address the usability issue for process algebra along two different directions. On the modeling side, we provide a set of guidelines inspired by the software architecture field, which should enforce a clear component-oriented approach to the process algebraic design of system families. On the verification side, we propose a component-oriented technique based on equivalence checking for the detection of architecture-level mismatches and the provision of related diagnostic information. Such a technique extends previous results in terms of generality of the considered mismatches, generality of the considered system topologies, and scalability to system families.

Key words: process algebra, usability, architectural description languages, component-oriented verification and diagnosis, equivalence checking.

1 Introduction

Process algebra [20,15,4,11,5] is a very rich theory that underpins the semantics of concurrent programming as well as the understanding of the behavior of communicating concurrent systems and their various aspects – mobility, performability, real-time constraints, and security. Process algebra relies on a small set of powerful operators – among which is a parallel composition operator – that inherently support compositionality, i.e. the ability to build process terms from smaller ones. Its semantics is formally defined

* Corresponding author: `bernardo@sti.uniurb.it`

through structural operational rules, which precisely establish for each process term the state transition graph that it stands for. Both syntax-oriented and semantics-oriented behavioral reasonings are possible on process terms via different equivalences, which capture different variants of the notion of same behavior possibly abstracting from unnecessary details.

Despite its strengths, after almost thirty years the use of process algebra outside the academia is still very limited. Due to some of its technicalities, process algebra is perceived by practitioners as being difficult to learn and use, which is further exacerbated by the short time-to-market constraint to which the information and communication technology industry is subject.

The limited usability of process algebra is the open problem that we address in this paper. We observe that there are at least two interrelated objectives to pursue. On the one hand, process algebra must be brought closer to the way we think of computing systems nowadays. Since they typically are component-based, it is necessary to support a friendly component-oriented way of modeling them, which hides the process algebraic technicalities. On the other hand, the use of process algebra needs to be integrated in the system development cycle. This requires understanding the appropriate phase or combination of phases of the cycle – requirement analysis, architectural design, component design, implementation, deployment, testing, and maintainance – in which process algebra can profitably be employed.

As a step towards the solution of the usability problem, we propose to revise process algebra in such a way that it can be employed in the architectural design of component-based systems [21,22]. Our proposal is conceptually divided into two parts. In the first part, we address the usability issue for process algebra from the modeling viewpoint. This is carried out by providing a set of guidelines – inspired by the architectural design level – that should enforce a component-oriented approach to the process algebraic design of system families, while hiding the technicalities of process algebra. The guidelines should be employed to turn process algebra into a fully fledged architectural description language (ADL), which elucidates the main architectural concepts behind the system design: components, connectors, and styles [22]. The guidelines are concerned with: (1) the separation of the behavior specification from the topology description, (2) the reuse of the specification of components and connectors, (3) the elicitation of the interactions, (4) the classification of the communications, (5) the combined use of textual and graphical notations, (6) the proper use of the process algebraic operators, and (7) the support for architectural styles, i.e. the design of system families. Although several process algebraic ADLs have appeared in the literature – like, e.g., Wright [3], Darwin/FSP [18,19], and PADL [8] – to the best of our knowledge this is the first time that the design rationale behind such languages is discussed in a detailed way and structured around a set of guidelines.

In the second part of our proposal, we address the usability issue from the analysis viewpoint. The aim is that of providing an analysis technique based on equivalence checking, which has the same component-oriented flavor as the syntax of a process algebraic ADL. The component orientation of such a technique should bring two advantages. First, the technique should be able to avoid computing the overall state space of a system description as much as possible. Second, in the case of property violation, the technique should be able to generate some diagnostic information through which the components and connectors responsible for the violation can be singled out and then modified to restore the property. Our contribution here is that the proposed technique extends the previous results developed in the software architecture field [3,17,16,12,8] in terms of generality of the considered properties, generality of the considered system topologies, and scalability to system families.

This paper, which is a full and revised version of [1,9,10] and builds on [8], is organized as follows. In Sect. 2 we present the guidelines to turn process algebra into an ADL for the specification of single systems. In Sect. 3 we discuss how to provide support for the design of system families within a process algebraic ADL. In Sect. 4 we present a technique based on equivalence checking for the component-oriented verification of mismatch freedom in process algebraic architectural descriptions of system families. Finally, in Sect. 5 we report some concluding remarks.

2 Turning Process Algebra into an ADL

In this section we discuss the first part of our proposal for enhancing the usability of process algebra. After introducing a simple system, which will be used as a running example throughout this section, and the typical process algebraic operators, we provide a set of guidelines inspired by the architectural design level. Such guidelines should enforce a component-oriented approach to the process algebraic design of systems, while hiding the technicalities of process algebra. In this section we present the guidelines (1) to (6). Guideline (7), which is related to the design of system families, will be presented in Sect. 3.

2.1 The Running Example

The guidelines will be illustrated by means of a running example based on a pipe-filter system. This system is composed of four identical filters and one pipe. Each filter stores the items it receives from the outside into a buffer of capacity ten, then processes the items in the order of their arrival and sends them out. Each filter is subject to failures and subsequent repairs. The four

filters are divided into one upstream filter and three downstream filters. Every item processed by the upstream filter is accepted by the pipe, which then forwards it to one of the three downstream filters according to the availability of free positions in their buffers. If more than one downstream filter has free positions, the choice is resolved nondeterministically.

2.2 Process Algebra Syntax

Let us consider a value-passing process algebra where the system descriptions are sequences of possibly recursive behavioral equations of the following form:

$$\mathtt{A}(\mathit{formal\_par\_list};\ \mathit{local\_var\_list}) = \mathtt{E}$$

where process term E has the following syntax:

$$\begin{aligned}
\mathtt{E} ::= {} & \mathtt{stop} \\
| \ & \mathtt{A}(\mathit{actual_par_list}) \\
| \ & \mathtt{a}.\mathtt{E} \quad | \quad \mathtt{a}?(\mathit{var_list}).\mathtt{E} \quad | \quad \mathtt{a}!(\mathit{expr_list}).\mathtt{E} \\
| \ & \mathtt{choice}\{\mathit{process_term_list}\} \\
| \ & \mathtt{cond}(\mathit{expr}) \mathrel{->} \mathtt{E} \\
| \ & \mathtt{E}/\mathtt{L} \\
| \ & \mathtt{E}\backslash\mathtt{H} \\
| \ & \mathtt{E}[\varphi] \\
| \ & \mathtt{E} \parallel_{\mathtt{S}} \mathtt{E}
\end{aligned}$$

A process term can be built out of the most frequently used static operators – inaction, behavior invocation, unstructured/input/output action prefix, choice, and conditional – and dynamic operators – hiding, restriction, relabeling, and multiway parallel composition. The admitted data types for formal parameters, local variables, and expressions are `boolean`, `integer`, bounded integer (`integer(min..max)`), and `real`. The admitted type constructors are `list`, `array`, and `record`. We assume that the semantics is defined in the usual interleaving style.

The pipe-filter system introduced in Sect. 2.1 can be specified in the process algebra above through the following behavioral equation:

$$\begin{aligned}\texttt{Pipe\_Filter} = \texttt{Upstream\_Filter}(0) \,\|_{\{\texttt{process\_accept\_item}\}} \\ \texttt{Pipe} \,\|_{\{\texttt{forward\_store\_item\_1}\}} \\ \texttt{Downstream\_Filter\_1}(0) \,\|_{\{\texttt{forward\_store\_item\_2}\}} \\ \texttt{Downstream\_Filter\_2}(0) \,\|_{\{\texttt{forward\_store\_item\_3}\}} \\ \texttt{Downstream\_Filter\_3}(0)\end{aligned}$$

where:

```
Upstream_Filter(integer(0..10) item_num; void) =
  choice {
    cond(item_num < buffer_size) ->
      store_item . Upstream_Filter(item_num + 1),
    cond(item_num > 0) ->
      process_accept_item . Upstream_Filter(item_num - 1),
    fail . repair . Upstream_Filter(item_num)
  }

Pipe(void; void) =
  process_accept_item . choice {
                          forward_store_item_1 . Pipe(),
                          forward_store_item_2 . Pipe(),
                          forward_store_item_3 . Pipe()
                        }

Downstream_Filter_1(integer(0..10) item_num; void) =
  choice {
    cond(item_num < buffer_size) ->
      forward_store_item_1 . Downstream_Filter_1(item_num + 1),
    cond(item_num > 0) ->
      process_item . Downstream_Filter_1(item_num - 1),
    fail . repair . Downstream_Filter_1(item_num)
  }

Downstream_Filter_2(integer(0..10) item_num; void) =
  choice {
    cond(item_num < buffer_size) ->
      forward_store_item_2 . Downstream_Filter_2(item_num + 1),
    cond(item_num > 0) ->
      process_item . Downstream_Filter_2(item_num - 1),
    fail . repair . Downstream_Filter_2(item_num)
  }
```

```
Downstream_Filter_3(integer(0..10) item_num; void) =
  choice {
    cond(item_num < buffer_size) ->
      forward_store_item_3 . Downstream_Filter_3(item_num + 1),
    cond(item_num > 0) ->
      process_item . Downstream_Filter_3(item_num - 1),
    fail . repair . Downstream_Filter_3(item_num)
  }
```

2.3 Guideline 1: Separating Behavior from Topology

Within a process algebraic description, both the system behavior and the system topology are expressed through occurrences of the parallel composition operator. Unfortunately, this makes the process algebraic description quite complicated in the presence of numerous components and connectors, as it becomes hard to understand which components and connectors are involved in which communications. The usability of process algebra can thus be improved through a clear separation of the system behavior from the system topology.

ARCHI_TYPE	⟨system name and data parameters⟩
ARCHI_ELEM_TYPES	⟨architectural element types⟩
ARCHI_TOPOLOGY	⟨architectural topology⟩
BEHAV_VARIATIONS	⟨possible behavioral variations⟩
END	

Table 1
Structure of a revised process algebraic description

This can be achieved by revising the structure of a process algebraic description, going from a flat sequence of behavioral equations to three different sections as shown in Table 1. The revised description starts with the name of the system – which we call an architectural type – and its data parameters. In the case of the pipe-filter system of Sect. 2.1, the header is as follows:

```
ARCHI_TYPE Pipe_Filter(const integer pf_buffer_size := 10,
                       const integer(0..pf_buffer_size)
                               pf_init_item_num := 0)
```

All the data parameters of an architectural type are constant identifiers that can be used throughout the rest of the description. For this reason, unlike

the formal parameters and local variables of the behavioral equations, their declaration is preceded by `const`.

After this header, the first section of the revised description should define the behavior of the components and connectors that form the system. For the sake of simplicity, from now on components and connectors will generically be termed architectural elements. The second section should define the topology of the system based on its architectural elements and their communication structure. Finally, the third section should provide some flexibility by allowing some parts of the system behavior to be changed according to some specific modeling- or analysis-related needs. As an example, behavioral variations may be concerned with making some system activities unobservable, preventing some activities from occurring, or renaming some activities.

2.4 Guideline 2: Specification Reuse

It might be the case that a process algebraic description contains several behavioral equations that differ only for the name of some actions, typically because the choice of the name of the actions is not free. In order to increase the degree of specification reuse within a system description – which reduces the time to prepare the description itself – we distinguish between the definition of an architectural element and the instantiation of an architectural element.

The definition of an architectural element, which we call an architectural element type (AET), takes place only once in the first section of the revised process algebraic description of Table 1. In the case of the pipe-filter system of Sect. 2.1, we observe that the behavioral equations for the four filters shown in Sect. 2.2 can be merged into a single AET, whose action names can be freely chosen:

```
ARCHI_ELEM_TYPES

  ELEM_TYPE Filter_Type(const integer buffer_size,
                        const integer(0..buffer_size)
                                init_item_num)

    BEHAVIOR
      Filter(integer(0..buffer_size) item_num := init_item_num;
             void) =
        choice {
          cond(item_num < buffer_size) ->
            store_item . Filter(item_num + 1),
```

```
          cond(item_num > 0) ->
            process_item . Filter(item_num - 1),
          fail . repair . Filter(item_num)
        }

  ELEM_TYPE Pipe_Type(void)

    BEHAVIOR
      Pipe(void; void) =
        accept_item . choice {
                        forward_item_1 . Pipe(),
                        forward_item_2 . Pipe(),
                        forward_item_3 . Pipe()
                      }
```

The instantiation of an architectural element, which we call an architectural element instance (AEI), takes place in the second section of the description. A single AET can have several AEIs, possibly equipped with different actual data parameters. In the case of the pipe-filter system, `Filter_Type` has four instances:

```
ARCHI_TOPOLOGY

  ARCHI_ELEM_INSTANCES
    F_0 : Filter_Type(pf_buffer_size, pf_init_item_num);
    P   : Pipe_Type();
    F_1 : Filter_Type(pf_buffer_size, pf_init_item_num);
    F_2 : Filter_Type(pf_buffer_size, pf_init_item_num);
    F_3 : Filter_Type(pf_buffer_size, pf_init_item_num)
```

2.5 Guideline 3: Interaction Elicitation

The actions occurring in the behavioral equations of an AET do not play the same role from the communication viewpoint. First of all, we distinguish between internal actions, which model activities local to the AET, and interactions, which are the interfaces through which the instances of the AET communicate with the rest of the system. The interactions are then subject to two further classifications. The first one divides the interactions into input and output interactions, depending on whether the AET is passive or active with respect to them. The second one divides the interactions into architectural interactions, which act as interfaces for the whole system, and local interactions,

whose scope remains within the system.

In order to enhance the readability of a process algebraic specification, we need to provide a way to explicitly elicit and classify the interactions in a component-oriented way. This is accomplished in two steps. In the first section of the revised process algebraic description of Table 1, the definition of each AET is completed with the declaration of its input and output interactions. In the second section, the architectural interactions are declared, while the local interactions are attached to each other in order to define the system communication structure.

As an example, observed that `fail` and `repair` are internal actions, the definition of `Filter_Type` provided in Sect. 2.4 must be completed as follows:

```
INPUT_INTERACTIONS  store_item
OUTPUT_INTERACTIONS process_item
```

while the definition of `Pipe_Type` must be completed as follows:

```
INPUT_INTERACTIONS  accept_item
OUTPUT_INTERACTIONS forward_item_1;
                    forward_item_2;
                    forward_item_3
```

Since the input interaction of the upstream filter and the output interactions of the three downstream filters are the entry and exit points of the overall system, while all the other filter interactions are used to exchange items through the pipe, the architectural topology of Sect. 2.4 must be completed as follows:

```
ARCHI_INTERACTIONS
  F_0.store_item;
  F_1.process_item; F_2.process_item; F_3.process_item

ARCHI_ATTACHMENTS
  FROM F_0.process_item TO P.accept_item;
  FROM P.forward_item_1 TO F_1.store_item;
  FROM P.forward_item_2 TO F_2.store_item;
  FROM P.forward_item_3 TO F_3.store_item

END
```

Some remarks are now in order. First, the dot notation is used to disambiguate among interactions with the same name belonging to different AEIs. Second,

the declaration of architectural interactions provides support for hierarchical modeling, as we shall see in more detail in Sect. 3. Third, local interactions that have to synchronize do not need to have the same name, thus increasing the readability of the description. Fourth, some static checks can easily be applied to the architectural topology, which permit to catch some modeling errors that would probably go undetected in traditional process algebraic descriptions. For instance, every local interaction must be involved in at least one attachment, while the internal actions and the architectural interactions cannot be involved in any attachment. As another example, every attachment must connect an output local interaction of an AEI to an input local interaction of another AEI.

2.6 Guideline 4: Communication Classification

As far as the interactions are concerned, it is worth noting that they may be involved in different forms of communication. Among the most frequently recurring ones, we mention one-to-one communications and conjuctive/disjunctive one-to-many communications. In order to account for them in a component-oriented way, in the first section of the revised process algebraic description of Table 1 the input and output interactions are further classified as uni-interactions, and-interactions, and or-interactions. A uni-interaction can be attached only to one interaction, while an and-interaction and an or-interaction can be attached to several uni-interactions. In the case of execution of an and-interaction, it synchronizes with all the uni-interactions attached to it (broadcast-like communication). In the case of execution of an or-interaction, instead, it synchronizes with only one of the uni-interactions attached to it, which is selected nondeterministically (client-server communication).

As an example, the declaration of the interactions of `Filter_Type` provided in Sect. 2.5 must be refined as follows:

```
INPUT_INTERACTIONS  UNI store_item
OUTPUT_INTERACTIONS UNI process_item
```

For `Pipe_Type` we observe that in general it may have to be connected with an arbitrary number of downstream filters, communicating with only one of them at a time. This can easily be achieved through an or-interaction after redefining `Pipe_Type` in a more concise way as follows:

```
ELEM_TYPE Pipe_Type(void)

  BEHAVIOR
    Pipe(void; void) = accept_item . forward_item . Pipe()
```

```
INPUT_INTERACTIONS  UNI accept_item
OUTPUT_INTERACTIONS OR  forward_item
```

The attachments are consequently modified as follows:

```
ARCHI_ATTACHMENTS
  FROM F_0.process_item TO P.accept_item;
  FROM P.forward_item   TO F_1.store_item;
  FROM P.forward_item   TO F_2.store_item;
  FROM P.forward_item   TO F_3.store_item
```

We conclude by noting that these additional communication-related qualifiers for the interactions provide support for further static checks of the architectural topology. For instance, every local uni-interaction of an AEI must be attached only to one local interaction of another AEI. As another example, every and- and or-interaction of an AEI must be attached only to uni-interactions of other AEIs different from each other.

2.7 Guideline 5: Graphical Notation

A pure textual notation can become cumbersome when modeling complex systems. While a textual notation is adequate for the description of the system behavior, a graphical notation can be more helpful for the definition of the system topology. We therefore combine the enhanced process algebraic notation introduced before with a graphical notation, inspired by flow graphs [20], to provide a visual aid. In an enriched flow graph, the AEIs are depicted as boxes, the local interactions are depicted as small black circles on the box border, the architectural interactions are depicted as small white squares on the box border, and the attachments are depicted as directed edges among local interactions. In the case of an and-interaction, the related small circle/square is extended with a triangle outside the box. In the case of an or-interaction, such a triangle is marked with a bar. We report in Fig. 1 the enriched flow graph for the pipe-filter system of Sect. 2.1.

2.8 Guideline 6: Transparent Use of the Static Operators

In the structure of the revised process algebraic description of Table 1, the use of the process algebraic operators is confined to the first section. In order to simplify the modeling process, the AETs are viewed as sequential entities,

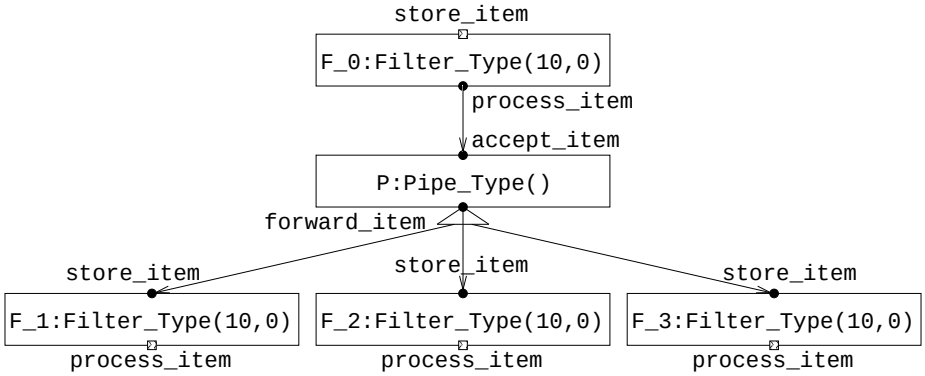

Fig. 1. Graphical description of `Pipe_Filter`

hence only the (easier) dynamic operators are at the disposal of the designer for the definition of the AET behavior. The (harder) static operators should not be explicitly used in a process algebraic architectural description. However, they play a fundamental role whenever the semantics of a process algebraic architectural description is given by translation into pure process algebra. In other words, the static operators should only be used in a way that is completely transparent, so that their technicalities do not burden the system description.

The translation semantics proceeds in two steps. In the first step, the focus is on the semantics of each AEI, which is defined to be the behavior of the corresponding AET, with the formal data parameters replaced by the corresponding actual data parameters. If the AET contains some or-interactions, each of them must be rewritten into a choice among as many indexed instances of a fresh uni-interaction as there are attachments involving the original or-interaction, in order to reflect the fact that each or-interaction can result in several distinct synchronizations.

Definition 2.1 Let $\mathcal{A}$ be an architectural type, let $\mathcal{C}$ be one of its AETs with formal data parameters $\mathtt{fp}_1, \ldots, \mathtt{fp}_m$ and behavior given by the list of behavioral equations $\mathcal{B}$, and let C be an instance of $\mathcal{C}$ with actual data parameters $\mathtt{ap}_1, \ldots, \mathtt{ap}_m$. The semantics of C is defined by

$$[\![C]\!] = \mathit{or\text{-}rewrite}(\mathcal{B})\{\mathtt{ap}_1/\mathtt{fp}_1, \ldots, \mathtt{ap}_m/\mathtt{fp}_m\}$$

where the curly braces enclose a syntactical substitution and $\mathit{or\text{-}rewrite}(\mathcal{B})$ is defined by structural induction on the process terms forming the right-hand side of the behavioral equations in $\mathcal{B}$ as follows:

$$
\begin{aligned}
\textit{or-rewrite}(\texttt{stop}) &= \texttt{stop} \\
\textit{or-rewrite}(\texttt{a.E}) &= \begin{cases} \texttt{a}.\textit{or-rewrite}(\texttt{E}) \\ \qquad \text{if a is not an or–interaction} \\ \texttt{choice}_{\texttt{i}=1}^{\texttt{n}}\{\texttt{a}_\texttt{i}.\textit{or-rewrite}(\texttt{E})\} \\ \qquad \text{if a is an or–interaction} \\ \qquad \text{involved in n attachments} \end{cases} \\
\textit{or-rewrite}(\texttt{choice}_{\texttt{i}=1}^{\texttt{k}}\{\texttt{E}_\texttt{i}\}) &= \texttt{choice}_{\texttt{i}=1}^{\texttt{k}}\{\textit{or-rewrite}(\texttt{E}_\texttt{i})\} \\
\textit{or-rewrite}(\texttt{A}(\textit{actual\_par\_list})) &= \texttt{A}(\textit{actual\_par\_list})
\end{aligned}
$$

■

For the pipe-filter system of Sect. 2.1 we have:

$$[\![\texttt{F\_0}]\!] = [\![\texttt{F\_1}]\!] = [\![\texttt{F\_2}]\!] = [\![\texttt{F\_3}]\!] = \texttt{Filter}\{\texttt{10/buffer\_size}, \texttt{0/item\_num}\}$$

$$[\![\texttt{P}]\!] = \textit{or-rewrite}(\texttt{Pipe})$$

where *or-rewrite*(Pipe) is given by the following behavioral equation for Pipe':

```
Pipe'(void; void) =
  accept_item . choice {
                  forward_item_1 . Pipe'(),
                  forward_item_2 . Pipe'(),
                  forward_item_3 . Pipe'()
                }
```

which is equivalent to the third behavioral equation of Sect. 2.2.

In the second step, the semantics of the whole description is derived by composing in parallel the semantics of its AEIs according to the specified attachments and by taking into account the possible behavioral variations. This is achieved by exploiting all the typical static operators: parallel composition, hiding, restriction, and relabeling. Since attached local interactions do not necessarily have the same name, while the CSP-like parallel composition operator that we use requires the synchronizing actions to have the same name, it is necessary to determine the number of fresh actions that are needed in order to make the AEIs interact according to the attachments. This requires to single out all the maximal sets of synchronizing local interactions, as all the members of a maximal set must be relabeled to the same fresh action. In the case of an attachment between two uni-interactions, the maximal set is composed of the two uni-interactions themselves. In the case of an and-interaction, we have a single maximal set composed of the and-interaction and all the uni-interactions attached to it. In the case of an or-interaction, we have as many maximal sets as there are attachments involving the or-interaction. Each of

such sets comprises one of the uni-interactions involved in the attachments and the corresponding uni-interaction obtained by indexing the or-interaction in the rewriting process (see Def. 2.1).

Given an architectural type $\mathcal{A}$, let $C_1, \ldots, C_n$ be some of its AEIs, and let i, j, k range over $\{1, \ldots, n\}$. For each AEI C_i, let $\mathcal{I}_{C_i} = \mathcal{LI}_{C_i} \cup \mathcal{AI}_{C_i}$ be the set of its local and architectural interactions, and $\mathcal{LI}_{C_i;C_1,\ldots,C_n} \subseteq \mathcal{LI}_{C_i}$ be the set of its local interactions attached to local interactions of $C_1, \ldots, C_n$. Once we have identified the maximal sets of synchronizing local interactions for the considered AEIs, we construct a set $\mathcal{S}(C_1, \ldots, C_n)$ composed of as many fresh actions as there are maximal sets of synchronizing local interactions. Then we relabel all the local interactions in the same set to the same fresh action. This is achieved by defining a set of injective relabeling functions of the form $\varphi_{C_i;C_1,\ldots,C_n} : \mathcal{LI}_{C_i;C_1,\ldots,C_n} \longrightarrow \mathcal{S}(C_1, \ldots, C_n)$ in such a way that $\varphi_{C_i;C_1,\ldots,C_n}(a_1) = \varphi_{C_j;C_1,\ldots,C_n}(a_2)$ iff $C_i.a_1$ and $C_j.a_2$ belong to the same set.

Definition 2.2 Let $\mathcal{A}$ be an architectural type and let $C_1, \ldots, C_n$ be some of its AEIs. The interacting semantics of C_i w.r.t. $C_1, \ldots, C_n$ is defined by

$$[\![C_i]\!]_{C_1,\ldots,C_n} = [\![C_i]\!][\varphi_{C_i;C_1,\ldots,C_n}]$$

■

Let us then define for each AEI and pair of AEIs in $C_1, \ldots, C_n$ the subset of fresh actions to which their local interactions are relabeled:

$$\mathcal{S}(C_i; C_1, \ldots, C_n) = \varphi_{C_i;C_1,\ldots,C_n}(\mathcal{LI}_{C_i;C_1,\ldots,C_n})$$
$$\mathcal{S}(C_i, C_j; C_1, \ldots, C_n) = \mathcal{S}(C_i; C_1, \ldots, C_n) \cap \mathcal{S}(C_j; C_1, \ldots, C_n)$$

Definition 2.3 Let $\mathcal{A}$ be an architectural type and let $C_1, \ldots, C_n$ be some of its AEIs. The interacting semantics of $C_1, \ldots, C_n$ is defined by

$$\begin{aligned}[\![C_1, \ldots, C_n]\!] = [\![C_1]\!]_{C_1,\ldots,C_n} \|_{\mathcal{S}(C_1,C_2;C_1,\ldots,C_n)} \\ [\![C_2]\!]_{C_1,\ldots,C_n} \|_{\mathcal{S}(C_1,C_3;C_1,\ldots,C_n) \cup \mathcal{S}(C_2,C_3;C_1,\ldots,C_n)} \cdots \\ \cdots \|_{\bigcup_{i=1}^{n-1} \mathcal{S}(C_i,C_n;C_1,\ldots,C_n)} [\![C_n]\!]_{C_1,\ldots,C_n}\end{aligned}$$

■

Definition 2.4 Let $\mathcal{A}$ be an architectural type, let $C_1, \ldots, C_n$ be all of its AEIs, let $\mathcal{H}$ be the set of its hidden actions, let $\mathcal{R}$ be the set of its restricted actions, and let φ be a relabeling function describing its renamings. The semantics of $\mathcal{A}$ before and after the behavioral variations are defined by

$$[\![\mathcal{A}]\!]_{\text{bbv}} = [\![C_1, \ldots, C_n]\!]$$
$$[\![\mathcal{A}]\!]_{\text{abv}} = [\![C_1, \ldots, C_n]\!] \,/\, \mathcal{H} \setminus \mathcal{R}\, [\varphi]$$

■

For the pipe-filter system of Sect. 2.1 we have:

$$
\begin{aligned}
&[\![\texttt{Pipe\_Filter}]\!]_{\mathrm{abv}} \;=\\
&\quad [\![\texttt{F\_0}]\!][\texttt{process\_item} \mapsto \texttt{F\_0.process\_item\#P.accept\_item}]\\
&\qquad \|_{\{\texttt{F\_0.process\_item\#P.accept\_item}\}}\\
&\quad [\![\texttt{P}]\!][\texttt{accept\_item} \mapsto \texttt{F\_0.process\_item\#P.accept\_item},\\
&\qquad \texttt{forward\_item\_1} \mapsto \texttt{P.forward\_item\_1\#F\_1.store\_item},\\
&\qquad \texttt{forward\_item\_2} \mapsto \texttt{P.forward\_item\_2\#F\_2.store\_item},\\
&\qquad \texttt{forward\_item\_3} \mapsto \texttt{P.forward\_item\_3\#F\_3.store\_item}]\\
&\qquad \|_{\{\texttt{P.forward\_item\_1\#F\_1.store\_item}\}}\\
&\quad [\![\texttt{F\_1}]\!][\texttt{accept\_item} \mapsto \texttt{P.forward\_item\_1\#F\_1.store\_item}]\\
&\qquad \|_{\{\texttt{P.forward\_item\_2\#F\_2.accept\_item}\}}\\
&\quad [\![\texttt{F\_2}]\!][\texttt{accept\_item} \mapsto \texttt{P.forward\_item\_2\#F\_2.store\_item}]\\
&\qquad \|_{\{\texttt{P.forward\_item\_3\#F\_3.accept\_item}\}}\\
&\quad [\![\texttt{F\_3}]\!][\texttt{accept\_item} \mapsto \texttt{P.forward\_item\_3\#F\_3.store\_item}]
\end{aligned}
$$

which is equivalent to the first behavioral equation of Sect. 2.2.

2.9 Comparison with Process Algebraic ADLs

Although several process algebraic ADLs have appeared in the literature – like, e.g., Wright [3], Darwin/FSP [18,19], and PADL [8] – as far as we know this is the first time that the design rationale behind such languages is discussed in a detailed way and structured around a set of guidelines.

The language that we have constructed here, which we call basic PADL, is an extension of the language presented in [8] from which the architectural invocation mechanism with behavioral parameter passing has temporarily been removed. Basic PADL is clearly inspired by Wright and Darwin/FSP. However, there are several differences that are worth mentioning. First, Wright distinguishes between components and connectors, while in basic PADL (similarly to Darwin/FSP) there are just architectural elements, each of which can be interpreted as being a component or a connector depending on the specific system. This avoids redundancy in the specifications caused by the presence of connectors whose behavior is trivial. Second, in Wright the description of each component/connector is accompanied by its ports/roles, whereas in basic PADL and in Darwin/FSP the interactions are simply expressed as actions. Since ports and roles can be retrieved whenever necessary through suitable applications of the hiding operator, avoiding ports and roles simplifies the

specifications. Third, Wright allows the connector behavior to be constrained through trace predicates, while this is not possible in basic PADL and in Darwin/FSP. As a consequence, in the last two languages the designer is forced to completely describe each behavior in a process algebraic way and then to verify that certain properties are satisfied by the behavior. This realizes a clear separation of concerns between modeling and analysis. Fourth, Darwin/FSP and an extension of Wright [2] allow systems with a dynamic architecture to be modeled, whereas this is not the case with basic PADL.

With respect to [8], basic PADL additionally provides support for the declaration of the forms of communication in which the interactions are involved (uni/and/or qualifiers) together with the related static checks, as well as support for the specification of behavioral variations. These features are present neither in Wright nor in Darwin/FSP.

2.10 Remarks on the Parallel Composition Operator

From a process algebraic viewpoint, the parallel composition operator is responsible for most of the modeling difficulties encountered by non-experts. The general problem with this operator is that it forces the system behavior and the system topology to be defined in an intertwined way. In this respect, it is worth mentioning that a performance-oriented variant of PADL called Æmilia, recently implemented in the software tool TwoTowers [6], is currently being used by some undergraduate and graduate students not necessarily familiar with formal methods. Such students, who were exposed to the previous version of TwoTowers based on pure stochastic process algebra, now feel much more confident about their ability of faithfully modeling the systems of their interest. They recognize that the reason for this is the replacement of the parallel composition operator with a section in which the system topology is separately described from the behavior by declaring the component and connector instances and the attachments between their interactions. Another recognized improvement is given by the various static checks that are automatically carried out based on the qualifiers associated with the elicited interactions.

Other problems with parallel composition are related to the specific operator that is used. The CCS-like parallel composition operator [20] is the simplest one, but supports only two-way synchronizations and requires the application of the restriction operator to enforce them. The CSP-like parallel composition operator [15] allows instead for multiway synchronizations without resorting to the restriction operator, but requires the synchronizing actions to have the same name and suffers from the fact that the synchronization sets depend on the order in which the process terms occur as operands. The ACP-like parallel composition operator [4] overcomes the two previous drawbacks through the

definition of a communication function on the set of actions, but still it is a binary operator.

A more friendly and expressive parallel composition operator is discussed in [13]. There it is proposed to adopt an n-ary parallel composition operator and to explicitly declare the interfaces of the involved process terms. Then the proposed operator is extended to deal with m-among-n synchronizations ($2 \leq m \leq n$), which take place when n process terms synchronize m by m on the same action. Similarly to [13], our approach suggests the component-oriented specification and attachment of the interactions in place of the definition of the synchronization sets, and provides support for n-among-n and 2-among-n synchronizations through and- and or-interactions, respectively. Our approach is thus less expressive than the one of [13], but resides at a higher level of abstraction, which in particular lets the designer free to choose different names for synchronizing interactions like in [4]. We may have adopted the parallel composition operator of [13] in the definition of the translation semantics, but we have preferred to avoid that in order not to complicate the proofs of the results of Sect. 4.

We conclude by mentioning the parallel composition operator recently proposed in [14]. This approach requires the specification of the sets of actions that can synchronize with each other and, within each set, the indication of those groups of actions that result in complete synchronizations. Although more flexible than [13], this approach cannot enforce m-among-n synchronizations because it assumes that complete synchronizations are closed with respect to set inclusion and it adopts a rule that favors the execution of the largest complete synchronizations.

3 Designing System Families

We now discuss the seventh guideline introduced in Sect. 1: providing support for the design of system families. This issue is closely related to the concept of architectural style [22], which denotes an organizational pattern that has been developed over the years, resulting in a family of systems having a common vocabulary of components and connectors as well as a common set of constraints on their topology. As examples of architectural styles we mention call-and-return systems (main program and subroutines, object-oriented programs, client-server systems, hierarchical layers), dataflow systems (pipe-filter systems), independent components (event-based systems), virtual machines (interpreters), and repositories (databases, hypertexts). From a practical viewpoint, the architectural styles should serve as a means to capitalize on codified principles and experience to specify, analyze, plan, and monitor the construction of complex systems with high levels of efficiency and confidence.

Unfortunately, it is hard to formalize the concept of architectural style, because the variability of the system behavior and of the system topology within an architectural style can be interpreted in different ways, with the interpretation possibly changing from style to style. In order to keep the task manageable, following [8] we advocate the use of an approximation of the concept of architectural style – called architectural type – which allows for a controlled variability of the system behavior and of the system topology. The controlled variability of the system behavior is achieved by allowing only the internal behavior of corresponding AETs to vary from instance to instance of an architectural type. As far as the controlled variability of the topology is concerned, we consider three kinds of topological extensions. The first one – exogenous extensions – permits to add further AEIs by attaching them to the architectural interactions, provided that the overall addendum complies with the original topology. The second one – endogenous extensions – is concerned with the introduction of further AEIs within the topology of the architectural type, in a way that respects the original topology. The third one – and/or extensions – allows the number of AEIs attached to and- or or-interactions to vary from instance to instance of an architectural type.

Technically speaking, the instances of an architectural type are obtained via an architectural invocation mechanism and can differ not only for their actual data parameters, but also for their actual behavior and topology. In this section we incrementally build on basic PADL by presenting the architectural invocation mechanism, then the passing of behavior-related architectural parameters, and finally the passing of topology-related architectural parameters resulting in exogenous, endogenous, and and/or extensions.

3.1 Architectural Invocation and Hierarchical Modeling

In its simplest form, an architectural invocation is expressed through the name of a previously defined architectural type, followed by its actual data parameters and the actual names for its architectural interactions enclosed in parentheses. The semantics of an architectural invocation is given by the semantics of the invoked architectural type (see Def. 2.4), to which a relabeling operator is applied in order to rename the architectural interactions to their actual names.

The architectural invocation mechanism opens the way to hierarchical modeling. This is achieved by allowing the behavior of an AET to be defined not only through a sequence of process algebraic behavioral equations, but also through an architectural invocation. Graphically, an AET defined through the invocation of an architectural type is represented as a box with a double border.

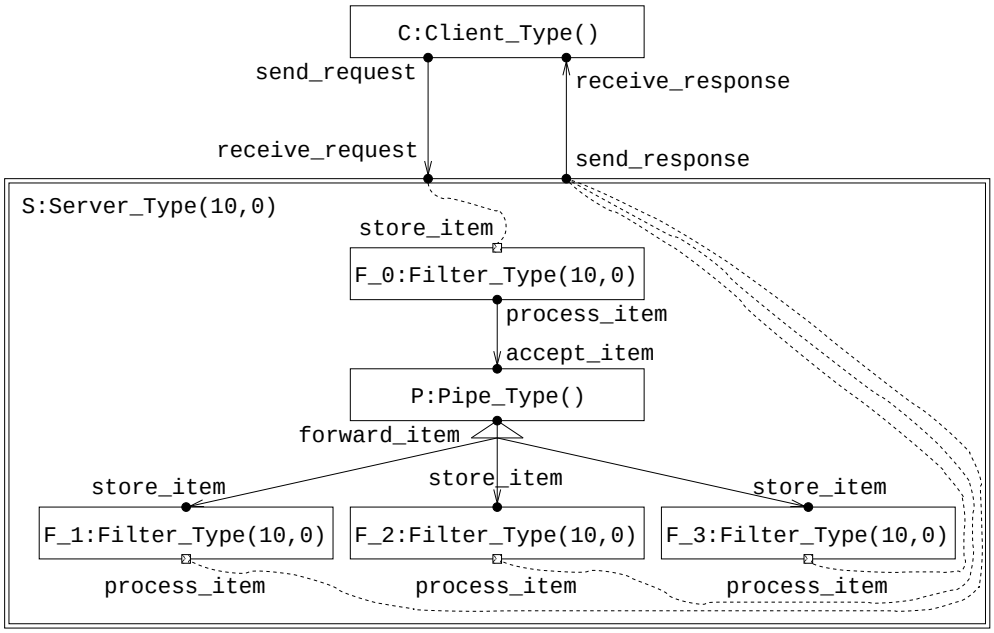

Fig. 2. Graphical description of `Client_Server`

As an example, suppose that we wish to describe a client-server system in which the server has the same structure and behavior as the pipe-filter system of Sect. 2.1. The graphical representation of the client-server system is shown in Fig. 2. As can be noted below, the support for hierarchical modeling is especially useful both to produce a more readable specification and to avoid defining the server from scratch:

```
ARCHI_TYPE Client_Server(const integer cs_buffer_size := 10,
                         const integer(0..cs_buffer_size)
                                 cs_init_item_num := 0)

ARCHI_ELEM_TYPES

  ELEM_TYPE Client_Type(void)

    BEHAVIOR
      Client(void; void) =
        send_request . receive_response . Client()

    INPUT_INTERACTIONS  UNI receive_response
    OUTPUT_INTERACTIONS UNI send_request

  ELEM_TYPE Server_Type(const integer buffer_size,
                        const integer(0..buffer_size)
                                init_item_num)
```

```
    BEHAVIOR
      Server(void; void) =
        Pipe_Filter(buffer_size, init_item_num;
                    receive_request, send_response,
                    send_response, send_response)

    INPUT_INTERACTIONS  UNI receive_request
    OUTPUT_INTERACTIONS UNI send_response

ARCHI_TOPOLOGY

  ARCHI_ELEM_INSTANCES
    C : Client_Type();
    S : Server_Type(cs_buffer_size, cs_init_item_num)

  ARCHI_INTERACTIONS
    void

  ARCHI_ATTACHMENTS
    FROM C.send_request  TO S.receive_request;
    FROM S.send_response TO C.receive_response

END
```

The behavior of `Server` is specified through an invocation of the previously defined architectural type `Pipe_Filter`. While `Pipe_Filter` has one architectural input uni-interaction (`F_0.store_item`) and three architectural output uni-interactions (`F_1.process_item`, `F_2.process_item`, `F_3.process_item`), `Server` has one local input uni-interaction (`receive_request`) and only one local output uni-interaction (`send_response`). The relation between the three architectural output uni-interactions of `Pipe_Filter` and the only local output uni-interaction of `Server` is established in the architectural invocation, where all the three architectural output uni-interactions are identically renamed to `send_response`.

3.2 Behavioral Architectural Parameters

The first set of architectural parameters that we add to the architectural invocation mechanism is composed of seven groups of behavioral architectural parameters that must be specified between the actual data parameters and the actual names for the architectural interactions. They represent the actual AETs, the actual AEIs, the actual architectural interactions, the actual attach-

ments, the actual hidings, the actual restrictions, and the actual renamings. The actual AETs are the actual types of components and connectors constituting the instance of the invoked architectural type. The presence of the other six behavioral architectural parameters is just a consequence of the fact that the actual AETs and their actions can have names that are different from the ones of the corresponding formal AETs and their actions, respectively. In other words, the AEIs, the architectural interactions, the attachments, the hidings, the restrictions, and the renamings in the definition of the invoked architectural type cannot always be reused in the invocation. Should reuse be possible for an entire group of behavioral parameters, these can be omitted from the architectural invocation.

We say that an actual AET behaviorally conforms to its corresponding formal AET if both AETs possess the same external behavior. The notion of same external behavior should be formalized through an equivalence that is: weak, in order to abstract from the internal actions; compositional with respect to the static operators, in order to provide support for lifting the notion of behavioral conformity from AETs to architectural types; fine enough, in order to preserve many properties of interest. To achieve these objectives, we adopt the weak bisimulation equivalence [20], which we denote by $\approx_{\mathrm{B}}$. Since an actual AET and the corresponding formal AET can give different names to corresponding interactions, the notion of same external behavior should abstract from these differences. This is accomplished by applying suitable injective relabeling functions to the interactions.

Definition 3.1 Let $\mathcal{C}_1, \mathcal{C}_2$ be two AETs with data parameter sets $\mathcal{D}_1, \mathcal{D}_2$, behaviors $\mathcal{B}_1, \mathcal{B}_2$, internal action sets $\mathcal{H}_1, \mathcal{H}_2$, and interaction sets $\mathcal{I}_1, \mathcal{I}_2$, respectively. Let the elements of $\mathcal{D}_1$ and $\mathcal{D}_2$ be consistent by number, order, and type. Let the elements of $\mathcal{I}_1$ and $\mathcal{I}_2$ be consistent by number, order, and qualifiers. We say that $\mathcal{C}_1$ behaviorally conforms to $\mathcal{C}_2$ iff there exist two injective relabeling functions φ_1, φ_2 for $\mathcal{I}_1, \mathcal{I}_2$, respectively, such that φ_1 and φ_2 have the same codomain and are qualifier-consistent and

$$\mathit{or\text{-}rewrite}(\mathcal{B}_1) \,/\, \mathcal{H}_1\,[\varphi_1] \approx_{\mathrm{B}} \mathit{or\text{-}rewrite}(\mathcal{B}_2) \,/\, \mathcal{H}_2\,[\varphi_2]$$

for all data value assignments to $\mathcal{D}_1$ and $\mathcal{D}_2$. ■

Definition 3.2 Let $\mathcal{A}_1$ be the definition of an architectural type and $\mathcal{A}_2$ be an architectural invocation of $\mathcal{A}_1$. We say that $\mathcal{A}_2$ behaviorally conforms to $\mathcal{A}_1$ iff they have:

- Data parameters that are consistent by number, order, and type.
- Behaviorally conformant AETs that are consistent by number and order.
- AEIs that are consistent by number, order, and type and have actual data parameters with the same values.
- Architectural interactions that are consistent by number, order, qualifiers, and membership to corresponding AEIs.

- Attachments that are consistent by number, order, and qualifiers and membership to corresponding AEIs of the involved local interactions. ■

If an architectural invocation behaviorally conforms to the definition of the invoked architectural type, then its semantics is built as in Def. 2.4, with the actual data parameters and the actual behavioral parameters replacing the corresponding formal parameters and the architectural interactions being relabeled with their actual names.

The following result, which is a slight reworking of a result shown in [8], is a straightforward consequence of Def. 3.2 and the fact that $\approx_{\mathrm{B}}$ is a congruence with respect to the static operators.

Theorem 3.3 Let $\mathcal{A}_1$ be the definition of an architectural type with internal action set $\mathcal{H}_1$ and interaction set $\mathcal{I}_1$ and let $\mathcal{A}_2$ be a behaviorally conformant architectural invocation of $\mathcal{A}_1$ with internal action set $\mathcal{H}_2$ and interaction set $\mathcal{I}_2$, where the architectural interactions occur with their actual names. If the values assigned to the formal data parameters of $\mathcal{A}_1$ are equal to the corresponding actual data parameters of $\mathcal{A}_2$, then there exist a relabeling function φ_1 for $\mathcal{I}_1$, which is injective at least on the local interactions, and an injective relabeling function φ_2 for $\mathcal{I}_2$, such that φ_1 and φ_2 have the same codomain and are qualifier-consistent and

$$\llbracket \mathcal{A}_1 \rrbracket_{\mathrm{bbv}} \,/\, \mathcal{H}_1\,[\varphi_1] \approx_{\mathrm{B}} \llbracket \mathcal{A}_2 \rrbracket_{\mathrm{bbv}} \,/\, \mathcal{H}_2\,[\varphi_2]$$

■

In the theorem above, φ_1 is not required to be injective over the architectural interactions because different architectural interactions in $\mathcal{I}_1$ may need to be put in correspondence with architectural interactions of $\mathcal{A}_2$ that are given the same actual name in $\mathcal{I}_2$ (see the architectural invocation of `Pipe_Filter` in Sect. 3.1).

Such a theorem provides us with an efficient algorithm for checking behavioral conformity. Instead of building the global state spaces for the architectural type definition and the architectural type invocation and checking them for weak bisimulation equivalence after applying suitable hidings and relabelings, we proceed as follows. For every pair of corresponding AETs without (resp. with) data parameters, we verify whether the local state spaces of such AETs (resp. of each pair of corresponding instances of such AETs) are weakly bisimulation equivalent after applying suitable hidings and relabelings. Therefore, the complexity of checking behavioral conformity at the architectural type level grows linearly with the number of AETs (resp. AEIs), instead of growing exponentially with the number of AEIs.

As an example, suppose the server of the architectural type `Client_Server` of Sect. 3.1 is a pipe-filter system in which the filters never fail:

```
ELEM_TYPE Perfect_Filter_Type(const integer buffer_size,
                              const integer(0..buffer_size)
                                      init_item_num)

  BEHAVIOR
    Perfect_Filter(integer(0..buffer_size) item_num :=
                                             init_item_num;
                   void) =
      choice {
        cond(item_num < buffer_size) ->
          store_item . Perfect_Filter(item_num + 1),
        cond(item_num > 0) ->
          process_item . Perfect_Filter(item_num - 1)
      }

  INPUT_INTERACTIONS  UNI store_item
  OUTPUT_INTERACTIONS UNI process_item
```

Then the architectural invocation defining the behavior of the server changes as follows:

```
Pipe_Filter(buffer_size, init_item_num;
            Perfect_Filter_Type, Pipe_Type;
            F_0 : Perfect_Filter_Type(buffer_size,
                                      init_item_num),
            P   : Pipe_Type(),
            F_1 : Perfect_Filter_Type(buffer_size,
                                      init_item_num),
            F_2 : Perfect_Filter_Type(buffer_size,
                                      init_item_num),
            F_3 : Perfect_Filter_Type(buffer_size,
                                      init_item_num);
            ;  /* reusing formal architectural interactions */
            ;  /* reusing formal attachments */
            ;  /* no hidings */
            ;  /* no restrictions */
            ;  /* no renamings */
            receive_request, send_response, send_response,
            send_response)
```

Since the semantics of each of the four instances of `Perfect_Filter_Type` is weakly bisimulation equivalent to the semantics of the corresponding instance of `Filter_Type` after hiding all the internal actions – `fail` and `repair` – and using the identical relabeling function, the above invocation of the architec-

tural type Pipe_Filter conforms to the definition of the same architectural type. As a consequence, the semantics of the above architectural invocation is weakly bisimulation equivalent to the semantics of the corresponding architectural definition, provided that the same internal actions as above are hidden and the identical relabeling function is applied.

3.3 Exogenous Extensions

Within an architectural type it is desirable to have some form of variability in the topology as well. As an example, consider the architectural type Pipe_Filter of Sect. 2.1. Every instance of such an architectural type admits a single pipe connector linking one upstream filter component to three downstream filter components. However, in principle it is reasonable to express by means of this architectural type any pipe-filter system with an arbitrary number of filter components and pipe connectors, such that every pipe connector links one upstream filter component to three downstream filter components. For instance, the enriched flow graph in Fig. 3 should be considered as a legal topological extension of the enriched flow graph in Fig. 1.

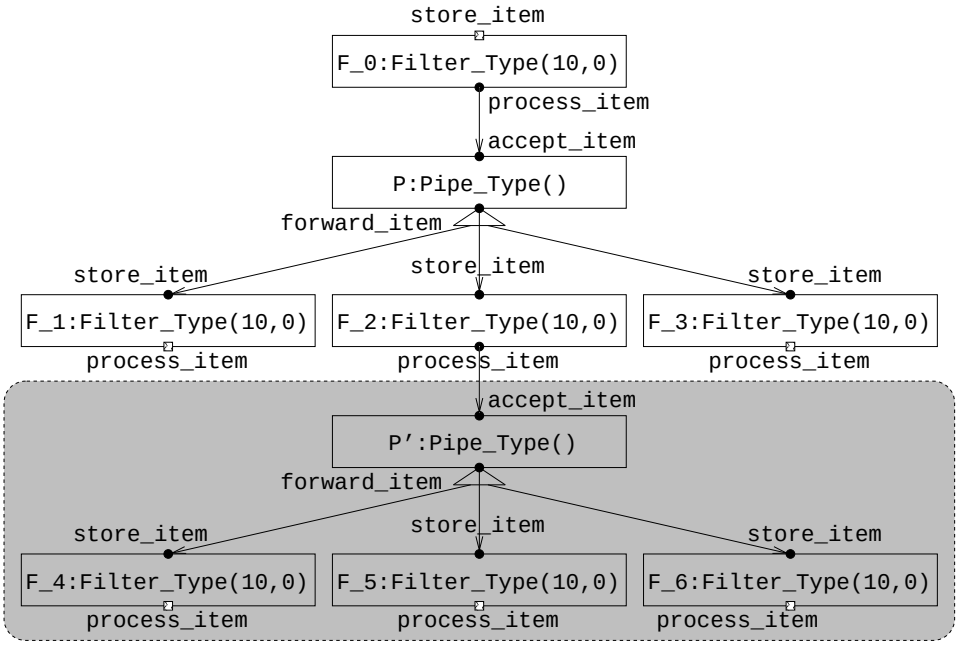

Fig. 3. Graphical description of an exogenous extension of Pipe_Filter

Since the architectural interactions are the frontier of an architectural type, the idea is to make it possible to extend an architectural type at some of its architectural interactions – exogenous extension – in a way that follows the topology prescribed by the architectural type itself. Note that this cannot be done at the local interactions because each of them must occur in at least one attachment, hence they are not free.

If present, the exogenous extensions are syntactically expressed within an architectural invocation between the actual attachments and the actual behavioral variations. A single exogenous extension is expressed through the keyword `EXO`, followed by four topological parameters enclosed in parentheses, which must satisfy certain constraints in order for the architectural invocation to topologically conform to the definition of the invoked architectural type.

The first topological parameter is a list of AEIs. Such AEIs represent the components and the connectors forming the exogenous extension. Each additional AEI must have a name different from the name of all the other actual AEIs in the architectural invocation, while its type must occur in the list of actual AETs of the architectural invocation, i.e. no new AET can be introduced in an exogenous extension.

The second topological parameter is a list of substitutions of architectural interactions of the additional AEIs for some of the actual architectural interactions. This parameter indicates all the (replacing) architectural interactions of the additional AEIs as well as all the (replaced) actual architectural interactions at which the exogenous extension takes place. Such replaced architectural interactions become local interactions within the exogenous extension, so each of them must be involved in at least one additional attachment. All the architectural interactions occurring in a substitution must belong to AEIs of the same type and have the same name. Every actual architectural interaction can occur only in one substitution of one exogenous extension.

The third topological parameter is a list of attachments. Such attachments describe the structure of the exogenous extension by connecting the additional AEIs to each other and to the replaced architectural interactions within the frontier of the topology of the invoked architectural type. Such attachments must follow the pattern prescribed by the topology of the invoked architectural type. Intuitively, it is not possible to introduce an additional attachment that has no correspondence in the topology of the invoked architectural type. Formally, this means that there must exist an injective function *corr* from the set of additional AEIs of the exogenous extension to the set of actual AEIs of the invoked architectural type, such that *corr* preserves the type and the value of the actual data parameters of corresponding AEIs, and for all interactions a of an arbitrary additional AEI C:

- $C.a$ is local/architectural iff $corr(C).a$ is local/architectural.
- There is an additional AEI C' with an additional attachment from $C.a$ (resp. $C'.a'$) to $C'.a'$ (resp. $C.a$) iff in the topology of the invoked architectural type there is an attachment from $corr(C).a$ (resp. $corr(C').a'$) to $corr(C').a'$ (resp. $corr(C).a$).
- There is an additional attachment from $C.a$ (resp. the replaced architectural interaction $K.b$) to the replaced architectural interaction $K.b$ (resp. $C.a$) iff

in the topology of the invoked architectural type there is an AEI K' of the same type as K with an attachment from $corr(C).a$ (resp. $K'.b$) to $K'.b$ (resp. $corr(C).a$).

The fourth topological parameter is a list of possible nested exogenous extensions.

In the presence of exogenous extensions, the possible actual behavioral variations can be concerned with the actions of the additional AEIs as well. We also observe that actual names must be provided only for the actual architectural interactions that are not replaced.

As an example of exogenous extension, let us consider the following invocation of the architectural type `Pipe_Filter` of Sect. 2.1 within the server of the architectural type `Client_Server` of Sect. 3.1, which results in the enriched flow graph of Fig. 3:

```
Pipe_Filter(buffer_size, init_item_num;
            Filter_Type, Pipe_Type;
            F_0 : Filter_Type(buffer_size, init_item_num),
            P   : Pipe_Type(),
            F_1 : Filter_Type(buffer_size, init_item_num),
            F_2 : Filter_Type(buffer_size, init_item_num),
            F_3 : Filter_Type(buffer_size, init_item_num);
            ;  /* reusing formal architectural interactions */
            ;  /* reusing formal attachments */
            EXO(P'  : Pipe_Type(),
                F_4 : Filter_Type(buffer_size,
                                  init_item_num),
                F_5 : Filter_Type(buffer_size,
                                  init_item_num),
                F_6 : Filter_Type(buffer_size,
                                  init_item_num);
                SUBST F_4.process_item,
                      F_5.process_item,
                      F_6.process_item
                  FOR F_2.process_item;
                FROM F_2.process_item TO P'.accept_item,
                FROM P'.forward_item  TO F_4.store_item,
                FROM P'.forward_item  TO F_5.store_item,
                FROM P'.forward_item  TO F_6.store_item;
                );  /* no nested exogenous extensions */
            ;  /* no hidings */
            ;  /* no restrictions */
            ;  /* no renamings */
```

```
receive_request, send_response, send_response,
send_response, send_response, send_response)
```

This exogenous extension occurs at one architectural output uni-interaction (`F_2.process_item`), which is replaced by three new architectural output uni-interactions (`F_4.process_item`, `F_5.process_item`, and `F_6.process_item`). As a consequence, in total we have one architectural input uni-interaction, which is renamed as `receive_request`, and five architectural output uni-interactions, each of which is renamed as `send_response`. Here we have that $corr(\mathtt{P}') = \mathtt{P}$, $corr(\mathtt{F\_4}) = \mathtt{F\_1}$, $corr(\mathtt{F\_5}) = \mathtt{F\_2}$, and $corr(\mathtt{F\_6}) = \mathtt{F\_3}$. The same kind of exogenous extension can take place at `F_1.process_item` or `F_3.process_item`, and can be nested within the exogenous extension itself at each of the three replacing architectural interactions.

An example of exogenous extension taking place at several architectural interactions instead of a single one is obtained if we modify the `Pipe_Filter` architectural type in such a way that the pipe connector has three upstream filter components and its `accept_item` action becomes an input or-interaction. In that case, all of `F_1.process_item`, `F_2.process_item`, and `F_3.process_item` must be replaced within the same substitution.

3.4 Endogenous Extensions

We now introduce another class of desirable topological extensions. Let us consider a ring of four stations, each adopting the same protocol – wait for a message from the previous station in the ring, process the received message, and send the processed message to the next station in the ring:

```
ARCHI_TYPE Station_Ring(void)

ARCHI_ELEM_TYPES

  ELEM_TYPE Init_Station_Type(void)

    BEHAVIOR
      Init_Station(void; void) =
        send_msg . receive_msg . process_msg . Init_Station()

    INPUT_INTERACTIONS  UNI receive_msg
    OUTPUT_INTERACTIONS UNI send_msg
```

```
  ELEM_TYPE Station_Type(void)

    BEHAVIOR
      Station(void; void) =
        receive_msg . process_msg . send_msg . Station()

    INPUT_INTERACTIONS  UNI receive_msg
    OUTPUT_INTERACTIONS UNI send_msg

ARCHI_TOPOLOGY

  ARCHI_ELEM_INSTANCES
    IS  : Init_Station_Type();
    S_1 : Station_Type();
    S_2 : Station_Type();
    S_3 : Station_Type()

  ARCHI_INTERACTIONS
    void

  ARCHI_ATTACHMENTS
    FROM IS.send_msg  TO S_1.receive_msg;
    FROM S_1.send_msg TO S_2.receive_msg;
    FROM S_2.send_msg TO S_3.receive_msg;
    FROM S_3.send_msg TO IS.receive_msg

END
```

Every instance of the architectural type `Station_Ring` admits a single initial station and three normal stations connected to form a ring, with the initial station being the first one allowed to send a message. However, it would be desirable to express by means of such an architectural type any ring system with an arbitrary number of normal stations. For instance, the enriched flow graph in Fig. 4 should be considered as a legal topological extension of the architectural type `Station_Ring`.

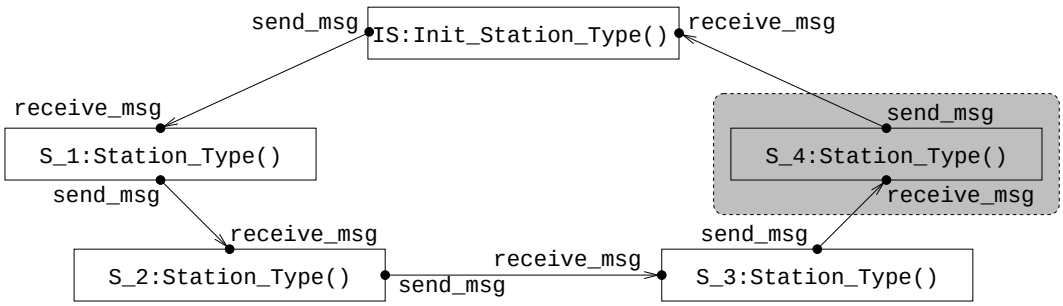

Fig. 4. Graphical description of an endogenous extension of `Station_Ring`

The idea behind this kind of topological extension, which we call endogenous, is that of introducing some additional AEIs within the topology itself in a controlled way. Unlike the exogenous extensions, an endogenous extension does not require passing topological parameters. Since an endogenous extension takes place within the topology of an architectural type, the definition of the architectural type has to be given in a way that permits the addition of AEIs of certain types in certain parts of the topology. This is easily achieved by describing the numbers of such AEIs through formal data parameters of the architectural type and by declaring some of the AEIs, of the architectural interactions, and of the attachments in an iterative way through an indexing mechanism controlled by the previous data parameters. As we shall see below, another difference between the endogenous extensions and the exogenous extensions is that the endogenous ones may introduce kinds of attachments that are not present in the instance of the considered architectural type with no additional AEIs, whereas this is not possible for the exogenous extensions.

As an example, we show below a different definition of the architectural type `Station_Ring` – where the identical parts are not repeated – which provides support for the desired endogenous extensions allowing for an arbitrary number of normal stations:

```
ARCHI_TYPE Station_Ring(const integer normal_station_num := 3)

ARCHI_ELEM_TYPES

  ELEM_TYPE Init_Station_Type(void)
    ...

  ELEM_TYPE Station_Type(void)
    ...

ARCHI_TOPOLOGY

  ARCHI_ELEM_INSTANCES
    IS : Init_Station_Type();
    FOR_ALL 1 <= i <= normal_station_num
      S[i] : Station_Type()

  ARCHI_INTERACTIONS
    void

  ARCHI_ATTACHMENTS
    FROM IS.send_msg TO S[1].receive_msg;
    FOR_ALL 1 <= i <= normal_station_num - 1
      FROM S[i].send_msg TO S[i + 1].receive_msg;
```

```
        FROM S[normal_station_num].send_msg TO IS.receive_msg

  END
```

For `normal_station_num` equal to 1 there is no attachment between two AEIs of type `Station_Type`, whereas attachments of this kind are introduced whenever `normal_station_num` is greater than 1. If `normal_station_num` is equal to 4 we get the endogenous extension depicted in Fig. 4.

3.5 And/Or Extensions

A special case of endogenous extension is the one that allows the number of AEIs of certain types attached to and- or or-interactions to vary in a controlled way. As an example, consider the architectural type `Pipe_Filter` of Sect. 2.1. Every instance of such an architectural type admits (via suitable exogenous extensions) a certain number of pipe connectors, each having an output or-interaction attached to the input uni-interaction of three downstream filter components. By means of the considered architectural type it would be desirable to express any pipe-filter system in which every pipe connector is attached to one upstream filter component and arbitrarily many downstream filter components. For instance, the enriched flow graph in Fig. 5 should be considered as a legal topological extension of the enriched flow graph in Fig. 1.

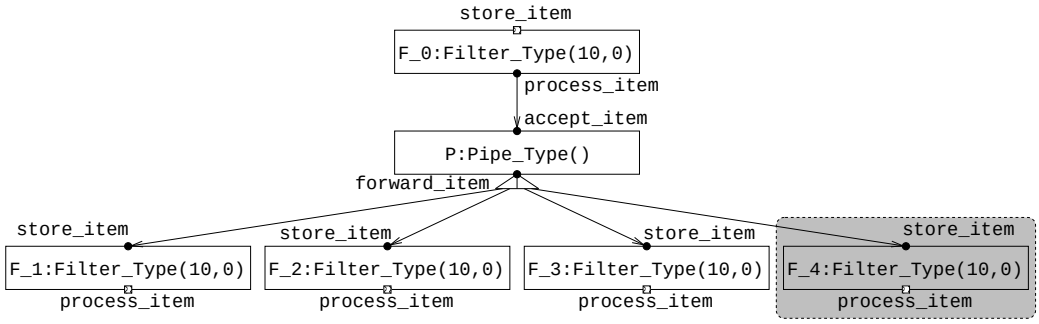

Fig. 5. Graphical description of an and/or extension of `Pipe_Filter`

Like the endogenous extensions, this kind of topological extension, which we call and/or, does not require the passing of architectural parameters. Instead, a formal data parameter is used to represent the number of AEIs attached to every extended and-/or-interaction, and an indexing mechanism controlled by such data parameters is employed to declare some of the AEIs, of the architectural interactions, and of the attachments in an iterative way. We observe that, in order for an and- or an or-interaction of an AEI to be extensible, whenever an AEI is attached to it with a uni-interaction, then the former AEI cannot be attached with uni-interactions to interactions of the latter AEI. If this were not the case, should an and/or extension take place then some uni-

interactions of the former AEI would be attached to several interactions of the latter AEI – as shown in Fig. 6 – which is not allowed. We also note that, unlike the endogenous extensions, the and/or extensions never introduce new kinds of attachments.

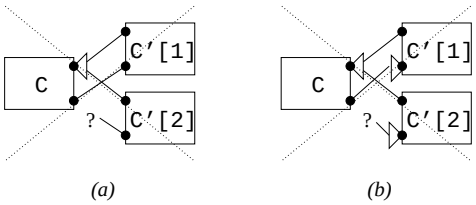

Fig. 6. Attachments preventing and/or extensions

As an example, we show below a different definition of the architectural type `Pipe_Filter` – where the identical parts are not repeated – which provides support for the desired and/or extensions allowing for an arbitrary number of downstream filter components:

```
ARCHI_TYPE Pipe_Filter(const integer pf_buffer_size := 10,
                       const integer(0..pf_buffer_size)
                               pf_init_item_num := 0,
                       const integer ds_filter_num := 4)

ARCHI_ELEM_TYPES

  ELEM_TYPE Filter_Type(const integer buffer_size,
                        const integer(0..buffer_size)
                                init_item_num)
    ...

  ELEM_TYPE Pipe_Type(void)
    ...

ARCHI_TOPOLOGY

  ARCHI_ELEM_INSTANCES
    F[0] : Filter_Type(pf_buffer_size, pf_init_item_num);
    P    : Pipe_Type();
    FOR_ALL 1 <= i <= ds_filter_num
      F[i] : Filter_Type(pf_buffer_size, pf_init_item_num)

  ARCHI_INTERACTIONS
    F[0].accept_item;
    FOR_ALL 1 <= i <= ds_filter_num
      F[i].process_item
```

```
    ARCHI_ATTACHMENTS
      FROM F[0].process_item TO P.accept_item;
      FOR_ALL 1 <= i <= ds_filter_num
        FROM P.forward_item TO F[i].accept_item

  END
```

For `ds_filter_num` equal to 4 we get the and/or extension depicted in Fig. 5, while for `ds_filter_num` equal to 3 we get the original architectural type depicted in Fig. 1.

3.6 Comparison with Process Algebraic ADLs

We call full PADL the language that we have constructed here. With respect to Wright and Darwin/FSP, the architectural invocation mechanism, the behavioral parameter passing, the exogenous extensions, the endogenous extensions, and the and/or extensions are new. With respect to [8], where the concepts of architectural invocation and behavioral parameter passing were originally proposed in a slightly different way, only the topological extensions are new.

4 Verifying Architectural Mismatch Freedom

The revision of the process algebra syntax leading to the construction of an ADL must be accompanied by an analogous effort on the verification side. Being able to reuse for a language like full PADL all the analysis machinery developed for process algebra is not enough. What is needed at the architectural design level is a technique to verify the freedom from architectural mismatches, i.e. those malfunctionings that arise when assembling together several components that are correct when considered in isolation. For efficiency and usefulness reasons, the verification technique mentioned above should be component oriented. This means that it should be able to infer the properties of the whole system from the properties of the system components and connectors, hopefully providing some diagnostic information in case of property violation useful to single out and modify the components and connectors responsible for the mismatches.

Apart from [12], where a generic notion of composability has been addressed, several results have been achieved via equivalence checking that are specifically concerned with deadlock freedom. In [3] a component-oriented deadlock freedom verification technique has been developed, which considers single pairs

of interactions of components and connectors communicating with each other. In [17,16] a more general technique has been proposed, which operates at the component level by taking into account the correlation among all the interactions of a component. In [8] an even more general technique has been presented, which considers not only the interactions between pairs of architectural elements, but also the interactions within sets of architectural elements forming a cycle.

As observed in the conclusion of [8], the proposed techniques suffer from some limitations concerned with the considered properties (only deadlock freedom), the distinguishing power of the employed equivalences (weak bisimulation equivalence and failure equivalence), the considered topologies (acyclic structures and single cycles), and the lack of full scalability of the mismatch freedom results from single systems to entire system families.

The main technical contribution of this paper is to generalize the previous results by showing that it is possible to verify in a component-oriented way an arbitrary property of a certain class over an entire architectural type having an arbitrary topology. After revising some technical definitions of [8], we first extend the sufficient condition of [8], which ensures deadlock freedom for a single architecture whose topology is acyclic or is a cycle, to the validity of an arbitrary property of a certain class for a single architecture with an arbitrary topology. Then we further extend this result from a single architecture to an architectural type, by considering behavioral parameter passing, exogenous extensions, endogenous extensions, and and/or extensions.

4.1 Class of Properties

The class of properties that we consider is characterized by three constraints. The first constraint is that such properties are expressed only in terms of local interactions. Unlike the internal actions and the architectural interactions, which can always be executed, the possibility or necessity of executing the local interactions and the order in which they are executed depend on the local interactions to which they are attached. Thus, architectural mismatches can only be generated by the wrong interplay of the local interactions of different AEIs. As a consequence of this constraint, the checks for verifying the absence of architectural mismatches that we shall introduce are applied locally to pairs of AEIs or cycles of AEIs by abstracting from their internal actions as well as their architectural interactions.

The second constraint is that, given a property $\mathcal{P}$ in the class, there exists a weak equivalence $\approx_{\mathcal{P}}$ coarser than $\approx_{\rm B}$ that preserves $\mathcal{P}$ – it never equates two process terms such that only one of them satisfies $\mathcal{P}$ – and is a congruence

with respect to the static operators of process algebra. The reason is that the previously mentioned checks, if passed, $\approx_{\mathcal{P}}$-equate a set of AEIs forming a star or a cycle to a single AEI in the set satisfying $\mathcal{P}$. Then this is compositionally exploited when scaling from the set of AEIs to the overall topology by replacing the considered set of AEIs with the $\approx_{\mathcal{P}}$-equivalent AEI satisfying $\mathcal{P}$. The fact that $\approx_{\mathcal{P}}$ is coarser than $\approx_{\rm B}$, used in the definition of behavioral conformity, is finally exploited when scaling from a single architecture to a family of behaviorally conformant architectural invocations.

The third constraint is that the (action-based) temporal logic in which the properties of the class are expressed does not allow the negation to be freely used. We shall admit the universal and existential path quantifiers, the temporal operators globally, eventually, and until, the modal operators weak diamond and weak box [1], the logical operators conjuction and disjunction, and the constant true. The reason is that, given a property $\mathcal{P}$ in the class, the application of the previously mentioned checks, if passed, results in $\approx_{\mathcal{P}}$-equating the architectural type under study in which some local interactions have been hidden to another architectural type satisfying $\mathcal{P}$. Therefore, to retrieve the validity of $\mathcal{P}$ for the original architectural type, in which the hidden local interactions are no longer hidden, we need to limit the way in which the negation can be used within $\mathcal{P}$.

From now on, when we mention an arbitrary property $\mathcal{P}$, it is understood that it belongs to the class of properties that we have just characterized.

4.2 Revisiting Compatibility and Interoperability

In the verification of architectural mismatch freedom, the presence/absence of cycles in the architectural topology plays a fundamental role. In this framework a cycle does not refer to a cycle in the enriched flow graph of the considered architectural type, but to a cycle in an enriched flow graph obtained from the original one by abstracting from the direction of the attachments and by collapsing all the attachments between two AEIs into a single attachment. We say that an architectural type is acyclic if its abstract enriched flow graph is acyclic.

The sufficient conditions that we shall introduce deal with checks to be carried out locally to pairs of AEIs or cycles of AEIs. Such checks will involve a variant of the interacting semantics of an AEI C_i with respect to a set of AEIs $C_1, \ldots, C_n$ (see Def. 2.2), in which all the actions of C_i that are not

[1] Each such modal operator is weak in the sense that, when checked against a state transition graph, it abstracts from all the transition labels different from the actions occurring in the modal operator itself.

local interactions attached to one of the other AEIs of the set are hidden. The checks will allow us to infer information about the interacting semantics of a set of AEIs or the semantics of an entire architectural type (before the behavioral variations), in which all the actions of the considered AEIs that are not local interactions attached to certain AEIs are hidden.

Definition 4.1 Let $\mathcal{A}$ be an architectural type, let $C_1, \ldots, C_n$ be some of its AEIs, and let Act be the set of all the actions. For all $1 \leq i \leq n$, the closed interacting semantics of C_i w.r.t. $C_1, \ldots, C_n$ is defined by

$$[\![C_i]\!]^{\mathrm{c}}_{C_1,\ldots,C_n} = [\![C_i]\!] \,/\, (Act - \mathcal{LI}_{C_i;C_1,\ldots,C_n})\,[\varphi_{C_i;C_1,\ldots,C_n}]$$

■

Definition 4.2 Let $\mathcal{A}$ be an architectural type, let $C_1, \ldots, C_n$ be some of its AEIs, and let $C'_1, \ldots, C'_{n'}$ be some of such AEIs. The closed interacting semantics of $C'_1, \ldots, C'_{n'}$ w.r.t. $C_1, \ldots, C_n$ is defined by

$$\begin{aligned}[\![C'_1, \ldots, C'_{n'}]\!]^{\mathrm{c}}_{C_1,\ldots,C_n} = {} & [\![C'_1]\!]^{\mathrm{c}}_{C_1,\ldots,C_n} \,\|_{\mathcal{S}(C'_1,C'_2;C'_1,\ldots,C'_{n'})} \\ & [\![C'_2]\!]^{\mathrm{c}}_{C_1,\ldots,C_n} \,\|_{\mathcal{S}(C'_1,C'_3;C'_1,\ldots,C'_{n'}) \cup \mathcal{S}(C'_2,C'_3;C'_1,\ldots,C'_{n'})} \cdots \\ & \cdots \,\|_{\cup_{i=1}^{n'-1} \mathcal{S}(C'_i,C'_{n'};C'_1,\ldots,C'_{n'})} [\![C'_{n'}]\!]^{\mathrm{c}}_{C_1,\ldots,C_n}\end{aligned}$$

■

Definition 4.3 Let $\mathcal{A}$ be an architectural type and let $C_1, \ldots, C_n$ be all of its AEIs. The closed semantics of $\mathcal{A}$ before the behavioral variations is defined by

$$[\![\mathcal{A}]\!]^{\mathrm{c}}_{\mathrm{bbv}} = [\![C_1, \ldots, C_n]\!]^{\mathrm{c}}_{C_1,\ldots,C_n}$$

■

In the following, a list of AEIs occurring in a subscript of a semantics or in a synchronization set can be shortened with the name of the set of such AEIs. In particular, if they are all the AEIs of an architectural type, their list can be replaced with the name of the architectural type.

As observed in [8], an acyclic architectural type can be viewed as the composition of several star topologies, each one being formed by an AEI K, called the center of the star topology, and a set of AEIs $C_1, \ldots, C_n$ attached to K, called the border of the star topology. The absence of cycles guarantees that $C_1, \ldots, C_n$ cannot directly communicate with each other. Therefore, in the acyclic case the validity of an arbitrary property $\mathcal{P}$ can be investigated by analyzing the interactions between the center K of the star topology and each of the AEIs constituting the border of the star topology. Intuitively, we say that K is compatible with C_i if the potential interactions of K with the AEIs in the border of the star topology are not altered when attaching C_i to K.

Definition 4.4 Let $\mathcal{A}$ be an architectural type and let K be one of its AEIs. The border of K is defined by

$$\mathcal{B}_K = \{C \mid C \text{ is attached to } K\}$$

■

Definition 4.5 Let $\mathcal{A}$ be an architectural type, let K be one of its AEIs, and let $\mathcal{B}_K = \{C_1, \ldots, C_n\}$. We say that K is $\mathcal{P}$-compatible with C_i iff

$$[\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{\mathcal{S}(K,C_i;K,\mathcal{B}_K)} [\![C_i]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \approx_{\mathcal{P}} [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}$$ ■

The result proven in [8] and illustrated on the specification of a compressing proxy system shows that, whenever K is deadlock free and compatible with every C_i (according to the weak bisimulation equivalence), then the whole star topology is deadlock free. It is easy to see that this holds for the validity of the arbitrary property $\mathcal{P}$. If the $\mathcal{P}$-compatibility check is not passed by some C_j, then this reveals – in a component-oriented way – a potential violation of $\mathcal{P}$ in the interaction between K and C_j. In [8] it is also proven that the absence of deadlock scales to the whole acyclic topology if the compatibility check is satisfied by every pair of attached AEIs. Once again, it is easy to see that this holds for the validity of the arbitrary property $\mathcal{P}$. We also note that, compared to checking $\mathcal{P}$ against the whole state space underlying an architectural type having an acyclic topology – whose complexity grows exponentially with the number of AEIs – the complexity of the application of the $\mathcal{P}$-compatibility check grows only linearly with the number of AEIs.

As further noted in [8], employing the peer-to-peer compatibility check described above is not enough in the case of a cyclic architectural type, as there may be additional causes of architectural mismatches due to the cyclic nature of the topology. In the cyclic case, to verify the validity of $\mathcal{P}$ we need a check that considers all the AEIs in the cycle, as each of them may interfere with the others. Intuitively, given a cycle $C_1, \ldots, C_n$, we say that C_i interoperates with the rest of the cycle if the potential interactions within the rest of the cycle are not altered when inserting C_i in the cycle, i.e. if the actual behavior of the rest of the cycle is the same as that expected by C_i.

Definition 4.6 Let $\mathcal{A}$ be an architectural type and let $C_1, \ldots, C_n$ be some of its AEIs forming a cycle in the abstract enriched flow graph of $\mathcal{A}$. We say that C_i $\mathcal{P}$-interoperates with the other AEIs in the cycle iff

$$[\![C_1, \ldots, C_n]\!]^{\mathrm{c}}_{C_1,\ldots,C_n} / (Act - \mathcal{S}(C_i; C_1, \ldots, C_n)) \approx_{\mathcal{P}} [\![C_i]\!]^{\mathrm{c}}_{C_1,\ldots,C_n}$$ ■

The result proven in [8] and illustrated on the specification of a cruise control system shows that, whenever there exists a deadlock free AEI in the cycle that interoperates with the other AEIs in the cycle (according to the weak bisimulation equivalence), then the whole cycle is deadlock free. It is easy to see that this holds for the validity of the arbitrary property $\mathcal{P}$. We observe that, although the worst-case complexity of the application of the $\mathcal{P}$-interoperability check grows exponentially with the number of AEIs in the cycle, in practice this can be mitigated if the state space of the overall cycle – in which some local interactions are hidden besides all the internal actions and all the architectural interactions – is built compositionally and minimized at each step

with respect to $\approx_{\mathcal{P}}$. The absence of a $\mathcal{P}$-interoperating AEI within the cycle reveals a potential violation of $\mathcal{P}$ within the cycle itself. Some component-based diagnostic information can be derived by applying the loop shrinking procedure proposed in [8]. In a generic shrinking step, we consider an AEI C_i in the cycle that does not $\mathcal{P}$-interoperate with the other AEIs in the cycle. The cause of a possible violation of $\mathcal{P}$ is within C_i, the rest of the cycle, or both. This can in principle be determined by considering the behavior of C_i together with the modal logic based diagnostic information coming from the failure of the $\mathcal{P}$-interoperability check. If we discover that a violation of $\mathcal{P}$ exists and that its source is within C_i, then we repair C_i and we repeat the $\mathcal{P}$-interoperability check, otherwise we shrink the cycle by replacing C_{i-1}, C_i, and C_{i+1} with a new AEI whose behavior is given by the parallel composition of the closed interacting semantics of the three original AEIs.

While the $\mathcal{P}$-compatibility check scales from single star topologies to arbitrary acyclic topologies, the $\mathcal{P}$-interoperability check does not scale from single cycles to arbitrary cyclic topologies. This is caused by subtle architectural mismatches that can arise from the interactions between intersecting cycles as well as between a cycle and an acyclic portion of the whole architectural topology. In particular, the $\mathcal{P}$-interoperability check applied to a cycle of AEIs $C_1, \ldots, C_n$ does not provide a sufficient condition for the satisfaction of $\mathcal{P}$ if some AEIs in the cycle interact with some other AEIs outside the cycle. In other words, if the frontier of the cycle is not empty, then the $\mathcal{P}$-interoperability check is not enough to decide the satisfaction of $\mathcal{P}$. Assume, e.g., that it is possible to find an AEI C_i in the cycle whose interactions are not affected by the behavior of the other AEIs in the cycle. Even if C_i $\mathcal{P}$-interoperates with the rest of the cycle, nothing can be deduced about the influence of other AEIs of the architectural topology upon the cycle in case some AEIs in the cycle interact with some AEIs outside the cycle. This is because, when checking $\mathcal{P}$-interoperability for C_i, we abstract from the interactions of the AEIs in the cycle that are attached to AEIs outside the cycle.

To overcome such a limitation, we now revise the previously defined notion of $\mathcal{P}$-interoperability by leaving all the local interactions of C_i visible. This is especially important when C_i is in the frontier of the cycle, i.e. whenever it is in the intersection of the cycle with other cycles or with an acyclic portion of the architectural topology. Below we also formalize the notion of frontier and the notion of cyclic border, which is a set of intersecting cycles.

Definition 4.7 Let $\mathcal{A}$ be an architectural type and let $C_1, \ldots, C_n$ be some of its AEIs forming a cycle in the abstract enriched flow graph of $\mathcal{A}$. We say that C_i $\mathcal{P}$-interoperates with the other AEIs in the cycle iff

$$[\![C_1, \ldots, C_n]\!]^{\mathrm{c}}_{\mathcal{A}} \,/\, (Act - \mathcal{S}(C_i; \mathcal{A})) \approx_{\mathcal{P}} [\![C_i]\!]^{\mathrm{c}}_{\mathcal{A}}$$

■

Definition 4.8 Let $\mathcal{A}$ be an architectural type and let $C_1, \ldots, C_n$ be some of

its AEIs. The frontier of $C_1, \ldots, C_n$ is defined by

$$\mathcal{F}_{C_1,\ldots,C_n} = \{C_i \in \{C_1, \ldots, C_n\} \mid \mathcal{LI}_{C_i;C_1,\ldots,C_n} \neq \mathcal{LI}_{C_i}\}$$ ■

Definition 4.9 Let $\mathcal{A}$ be an architectural type and let K be one of its AEIs belonging to some cycle in the abstract enriched flow graph of $\mathcal{A}$. The cyclic border of K is defined by

$$\mathcal{CB}_K = \{K\} \cup \{H \mid \exists C_1, \ldots, C_n.\, K, H, C_1, \ldots, C_n \text{ form a cycle}\}$$ ■

Definition 4.10 Let $\mathcal{A}$ be an architectural type and let K_1 and K_2 be two of its AEIs. We say that $\mathcal{CB}_{K_1}$ topologically conforms to $\mathcal{CB}_{K_2}$ iff there exists a bijection between them that preserves for the AEIs their type, their attachments within the cyclic border, and their membership to the frontier of the cyclic border. ■

4.3 Generalization to an Arbitrary Topology

The idea underlying the generalization of the results of [8] for an arbitrary property $\mathcal{P}$ to an arbitrary topology is to view an acyclic topology as a special topology to which every other topology can be reduced. Given an arbitrary topology that is not acyclic, from a conceptual viewpoint we may think of proceeding by reducing every cyclic portion of the topology satisfying the (revised) $\mathcal{P}$-interoperability check into a single $\approx_{\mathcal{P}}$-equivalent AEI, until we reach a point in which the $\mathcal{P}$-interoperability check does not succeed or we end up with an acyclic topology, to which we finally apply the $\mathcal{P}$-compatibility check. In practice, this reduction is implemented through a cycle covering strategy.

Definition 4.11 Let $\mathcal{A}$ be an architectural type. A cycle covering strategy σ for $\mathcal{A}$ is defined by the following algorithm:

(1) All the AEIs in the abstract enriched flow graph of $\mathcal{A}$ are initially unmarked.
(2) While there are unmarked AEIs in the cycles of the abstract enriched flow graph of $\mathcal{A}$:
  (a) Pick out one such AEI, say K.
  (b) Mark all the AEIs in $\mathcal{CB}_K$. ■

The application of a cycle covering strategy σ to a cyclic architectural type $\mathcal{A}$ results in a set of cyclic borders, denoted by $\mathcal{CB}_\sigma$, that involve every AEI belonging to a cycle in the abstract enriched flow graph of $\mathcal{A}$.

Lemma 4.12 Given a cyclic architectural type $\mathcal{A}$ and a cycle covering strategy σ for it that results in the set of cyclic borders $\mathcal{CB}_\sigma = \{\mathcal{CB}_{K_1}, \ldots, \mathcal{CB}_{K_n}\}$,

then for any pair of different cyclic borders $\mathcal{CB}_{K_i}$ and $\mathcal{CB}_{K_j}$ in $\mathcal{CB}_\sigma$, $\mathcal{CB}_{K_i}$ can be directly attached to $\mathcal{CB}_{K_j}$ in two different ways only:

(1) $\mathcal{CB}_{K_i}$ and $\mathcal{CB}_{K_j}$ interact through a single, shared AEI K.
(2) $\mathcal{CB}_{K_i}$ and $\mathcal{CB}_{K_j}$ do not share any AEI, but they interact through attachments between a single AEI H of $\mathcal{CB}_{K_i}$ and a single AEI H' of $\mathcal{CB}_{K_j}$.

Proof As far as condition (1) is concerned, assume that $\mathcal{CB}_{K_i}$ and $\mathcal{CB}_{K_j}$ share another AEI H. Then the abstract enriched flow graph of $\mathcal{A}$ would contain a cycle including K_i, K, K_j, and H, thus contradicting the hypothesis that $\mathcal{CB}_{K_i}$ is the cyclic border of K_i. Similarly, if there exists an attachment between an AEI H of $\mathcal{CB}_{K_i}$ and an AEI H' of $\mathcal{CB}_{K_j}$, then the abstract enriched flow graph of $\mathcal{A}$ would contain a cycle including K_i, K, K_j, H', and H, thus contradicting the hypothesis that $\mathcal{CB}_{K_i}$ is the cyclic border of K_i.

As far as condition (2) is concerned, assume that there exists another attachment between an AEI H'' of $\mathcal{CB}_{K_i}$ and an AEI H''' of $\mathcal{CB}_{K_j}$. Then the abstract enriched flow graph of $\mathcal{A}$ would contain a cycle including K_i, H, H', K_j, H''', and H'', thus contradicting the hypothesis that $\mathcal{CB}_{K_i}$ is the cyclic border of K_i. On the other hand, if there exists another attachment between an AEI H'' of $\mathcal{CB}_{K_i}$ and H', then the abstract enriched flow graph of $\mathcal{A}$ would contain a cycle including K_i, H, H', and H'', thus contradicting the hypothesis that $\mathcal{CB}_{K_i}$ and $\mathcal{CB}_{K_j}$ do not share any AEI. We can argue similarly in case of an attachment between an AEI H'' of $\mathcal{CB}_{K_j}$ and H. ■

Definition 4.13 Let $\mathcal{A}$ be an architectural type. A cycle covering strategy σ for $\mathcal{A}$ is said to be total iff, when replacing each cyclic border $\mathcal{CB}_{K_i} = \{H_1, \ldots, H_l\}$ in $\mathcal{CB}_\sigma$ with an AEI whose behavior is isomorphic to

$$[\![H_1, \ldots, H_l]\!]^{\mathrm{c}}_{\mathcal{A}} \,/\, (Act - \bigcup_{H_j \in \mathcal{F}_{H_1,\ldots,H_l}} \mathcal{S}(H_j; \mathcal{A}))$$

the obtained architectural topology is acyclic. ■

In the theorem below, a sufficient condition for the satisfaction of $\mathcal{P}$ is provided for an architectural type $\mathcal{A}$ with an arbitrary topology under three assumptions. First, every AEI of $\mathcal{A}$ must satisfy $\mathcal{P}$. Second, every AEI of $\mathcal{A}$ that belongs to an acyclic portion or to the frontier of some cycle in the abstract enriched flow graph of $\mathcal{A}$ must be $\mathcal{P}$-compatible with each AEI that is attached to it but does not belong to any of the cycles involving the former AEI. This ensures the satisfaction of $\mathcal{P}$ for acyclic portions of the topology. Third, if $\mathcal{A}$ has a cyclic topology, then there must exist a total cycle covering strategy for $\mathcal{A}$ such that two constraints are satisfied, which are concerned with cyclic borders. The first constraint requires that, if $\mathcal{A}$ is formed by a single cyclic border with empty frontier, then it must contain an AEI that $\mathcal{P}$-interoperates with the other AEIs in the cyclic border. The second constraint requires that every AEI in the frontier of any cyclic border must $\mathcal{P}$-interoperate with all

the other AEIs belonging to the cyclic border. This ensures a $\mathcal{P}$-compliant combination of cyclic and acyclic portions of the topology.

Theorem 4.14 Let $\mathcal{A}$ be an architectural type with an arbitrary topology. Suppose that the following conditions hold:

(1) For every AEI K of $\mathcal{A}$, $[\![K]\!]^{\mathrm{c}}_{\mathcal{A}}$ satisfies $\mathcal{P}$.
(2) For every AEI K that belongs to an acyclic portion or to the frontier of some cycle in the abstract enriched flow graph of $\mathcal{A}$, K is $\mathcal{P}$-compatible with each AEI in $\{C \in \mathcal{B}_K \mid C \notin \mathcal{CB}_K\}$.
(3) If $\mathcal{A}$ is cyclic, then there exists a set $\mathcal{CB}_\sigma$ of cyclic borders generated by a total cycle covering strategy σ such that:
   I. If $\mathcal{CB}_\sigma$ has a single cyclic border $\{C_1, \ldots, C_n\}$ such that $\mathcal{F}_{C_1,\ldots,C_n} = \emptyset$, then there exists an AEI C_i in the cyclic border that $\mathcal{P}$-interoperates with $C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n$.
   II. Otherwise, for each AEI C_i in the frontier of a cyclic border $\{C_1, \ldots, C_n\}$ in $\mathcal{CB}_\sigma$, C_i $\mathcal{P}$-interoperates with $C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n$.

Then $[\![\mathcal{A}]\!]^{\mathrm{c}}_{\mathrm{bbv}}$ satisfies $\mathcal{P}$.

Proof We proceed by induction on the number m of cycles in the abstract enriched flow graph of $\mathcal{A}$:

- If $m = 0$ then $\mathcal{A}$ is acyclic. To avoid trivial cases, suppose that there are at least two AEIs in $\mathcal{A}$. We preliminarily prove that, given an AEI K of $\mathcal{A}$ with $\mathcal{B}_K = \{C_1, \ldots, C_k\}$, $[\![K, C_1, \ldots, C_k]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \approx_{\mathcal{P}} [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}$ by proceeding by induction on the number k of AEIs in the border of K:
  · If $k = 1$ then $[\![K, C_1]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} = [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{\mathcal{S}(K,C_1;K,\mathcal{B}_K)} [\![C_1]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \approx_{\mathcal{P}} [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}$ by virtue of (2).
  · Let the result hold for a certain $k \geq 1$ and suppose that the border of K contains $k + 1$ AEIs. Then

$$
\begin{aligned}
&[\![K, C_1, \ldots, C_{k+1}]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \;= \\
&\quad [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{\mathcal{S}(K,C_1;K,\mathcal{B}_K)} \\
&\qquad [\![C_1]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{\mathcal{S}(K,C_2;K,\mathcal{B}_K) \cup \mathcal{S}(C_1,C_2;K,\mathcal{B}_K)} \\
&\qquad\quad [\![C_2]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{\mathcal{S}(K,C_3;K,\mathcal{B}_K) \cup \mathcal{S}(C_1,C_3;K,\mathcal{B}_K) \cup \mathcal{S}(C_2,C_3;K,\mathcal{B}_K)} \cdots \\
&\qquad\qquad \cdots \parallel_{\mathcal{S}(K,C_{k+1};K,\mathcal{B}_K) \cup \bigcup_{i=1}^{k} \mathcal{S}(C_i,C_{k+1};K,\mathcal{B}_K)} [\![C_{k+1}]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}
\end{aligned}
$$

  Due to the acyclicity of $\mathcal{A}$, we have that

$$\begin{aligned}
&[\![K, C_1, \ldots, C_{k+1}]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \;=\\
&\quad [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,\|_{\mathcal{S}(K,C_1;K,\mathcal{B}_K)}\\
&\qquad [\![C_1]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,\|_{\mathcal{S}(K,C_2;K,\mathcal{B}_K)}\\
&\qquad\quad [\![C_2]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,\|_{\mathcal{S}(K,C_3;K,\mathcal{B}_K)} \;\cdots\\
&\qquad\qquad \cdots \,\|_{\mathcal{S}(K,C_{k+1};K,\mathcal{B}_K)}\, [\![C_{k+1}]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}
\end{aligned}$$

By virtue of (2) applied to K and C_1 and the fact that $\approx_{\mathcal{P}}$ is a congruence with respect to the parallel composition operator, we have that

$$\begin{aligned}
&[\![K, C_1, \ldots, C_{k+1}]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \;\approx_{\mathcal{P}}\\
&\quad [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,\|_{\mathcal{S}(K,C_2;K,\mathcal{B}_K)}\\
&\qquad [\![C_2]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,\|_{\mathcal{S}(K,C_3;K,\mathcal{B}_K)} \;\cdots\\
&\qquad\quad \cdots \,\|_{\mathcal{S}(K,C_{k+1};K,\mathcal{B}_K)}\, [\![C_{k+1}]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}
\end{aligned}$$

from which the result follows by the induction hypothesis.

Now we prove that $[\![\mathcal{A}]\!]^{\mathrm{c}}_{\mathrm{bbv}}$ satisfies $\mathcal{P}$ by proceeding by induction on the number s of star topologies in the abstract enriched flow graph of $\mathcal{A}$:

- If $s = 1$ then, denoted by K the center of the only star topology in the abstract enriched flow graph of $\mathcal{A}$, we have that $[\![\mathcal{A}]\!]^{\mathrm{c}}_{\mathrm{bbv}} \approx_{\mathcal{P}} [\![K]\!]^{\mathrm{c}}_{\mathcal{A}}$ because of the previously proved result. The result then follows from (1) and the fact that $\approx_{\mathcal{P}}$ preserves $\mathcal{P}$.
- Let the result hold for a certain $s \geq 1$ and suppose that the abstract enriched flow graph of $\mathcal{A}$ is composed of $s + 1$ star topologies. Due to the acyclicity of $\mathcal{A}$, there must exist a star topology, say composed of the AEIs $K, H, C_1, \ldots, C_h$ and centered on K, that is attached – with one of the AEIs in its border, say H – to only one other star topology in the abstract enriched flow graph of $\mathcal{A}$. If we replace $K, C_1, \ldots, C_h$ with a single AEI K' whose behavior is isomorphic to $[\![K, C_1, \ldots, C_h]\!]^{\mathrm{c}}_{\mathcal{A}}$, then $[\![K']\!]^{\mathrm{c}}_{\mathcal{A}} \approx_{\mathcal{P}} [\![K]\!]^{\mathrm{c}}_{\mathcal{A}}$ because of the previously proved result. If we further replace K' and H with a single AEI H' whose behavior is isomorphic to $[\![H, K']\!]^{\mathrm{c}}_{\mathcal{A}} \,/\, (Act - \mathcal{S}(H; \mathcal{A}))$, then

$$\begin{aligned}
[\![H']\!]^{\mathrm{c}}_{\mathcal{A}} &\approx_{\mathcal{P}} [\![H]\!]^{\mathrm{c}}_{\mathcal{A}} \,\|_{\mathcal{S}(H,K';\mathcal{A})} ([\![K']\!]^{\mathrm{c}}_{\mathcal{A}} \,/\, (Act - \mathcal{S}(H; \mathcal{A})))\\
&\approx_{\mathcal{P}} [\![H]\!]^{\mathrm{c}}_{\mathcal{A}} \,\|_{\mathcal{S}(H,K';\mathcal{A})} [\![K']\!]^{\mathrm{c}}_{K,H}\\
&\approx_{\mathcal{P}} [\![H]\!]^{\mathrm{c}}_{\mathcal{A}} \,\|_{\mathcal{S}(H,K;\mathcal{A})} [\![K]\!]^{\mathrm{c}}_{K,H}\\
&\approx_{\mathcal{P}} [\![H]\!]^{\mathrm{c}}_{\mathcal{A}}
\end{aligned}$$

because of what previously proved, the fact that $\approx_{\mathcal{P}}$ is a congruence with respect to the parallel composition operator, and (2). In other words, if we replace the considered star topology with a single AEI whose behavior is isomorphic to the closed interacting semantics of its AEIs where only the interactions of H are left visible, then we obtain an architectural type $\mathcal{A}'$ satisfying (1), (2), and (3) and having an acyclic topology with one fewer

star topology. Then by the induction hypothesis it follows that $[\![\mathcal{A}']\!]^{\rm c}_{\rm bbv}$ satisfies $\mathcal{P}$. Since

$$[\![\mathcal{A}']\!]^{\rm c}_{\rm bbv} \approx_{\mathcal{P}} [\![\mathcal{A}]\!]^{\rm c}_{\rm bbv} \,/\, (\mathcal{S}(K;\mathcal{A}) - \mathcal{S}(H;\mathcal{A}))$$

we derive that $[\![\mathcal{A}]\!]^{\rm c}_{\rm bbv}$ satisfies $\mathcal{P}$, because $\approx_{\mathcal{P}}$ preserves $\mathcal{P}$ and $\mathcal{P}$ does not contain any free use of the negation.

- Let the result hold for a certain $m \geq 0$ and consider an architectural type $\mathcal{A}$ satisfying (1), (2), and (3), whose abstract enriched flow graph has $m+1$ cycles. Let $\mathcal{CB} = \{C_1, \ldots, C_n\}$ be a cyclic border in $\mathcal{CB}_\sigma$ that, by virtue of Def. 4.13, directly interacts with at most one cyclic border in $\mathcal{CB}_\sigma$. Now we replace the AEIs $C_1, \ldots, C_n$ with a new AEI C whose behavior is isomorphic to $[\![C_1, \ldots, C_n]\!]^{\rm c}_{\mathcal{A}} / (Act - \bigcup_{C_j \in \mathcal{F}_{C_1,\ldots,C_n}} \mathcal{S}(C_j;\mathcal{A}))$, thus obtaining an architectural type $\mathcal{A}'$ such that:

  · C preserves (1). In fact, by (3), there exists C_i such that

  $$[\![C_1, \ldots, C_n]\!]^{\rm c}_{\mathcal{A}} / (Act - \mathcal{S}(C_i;\mathcal{A})) \approx_{\mathcal{P}} [\![C_i]\!]^{\rm c}_{\mathcal{A}}$$

  from which we derive that $[\![C_1, \ldots, C_n]\!]^{\rm c}_{\mathcal{A}} / (Act - \mathcal{S}(C_i;\mathcal{A}))$ satisfies $\mathcal{P}$ because so does $[\![C_i]\!]^{\rm c}_{\mathcal{A}}$ due to (1) and $\approx_{\mathcal{P}}$ preserves $\mathcal{P}$. Therefore, we also have that $[\![C_1, \ldots, C_n]\!]^{\rm c}_{\mathcal{A}} / (Act - \bigcup_{C_j \in \mathcal{F}_{C_1,\ldots,C_n}} \mathcal{S}(C_j;\mathcal{A}))$ satisfies $\mathcal{P}$, because $\mathcal{P}$ does not contain any free use of the negation.

  · C preserves (2). In fact, let H be an AEI attached to C because it was previously attached to an AEI C_i of $\mathcal{F}_{C_1,\ldots,C_n}$. By (2) we have that

  $$[\![C_i]\!]^{\rm c}_{C_i,\mathcal{B}_{C_i}} \|_{\mathcal{S}(C_i,H;C_i,\mathcal{B}_{C_i})} [\![H]\!]^{\rm c}_{C_i,\mathcal{B}_{C_i}} \approx_{\mathcal{P}} [\![C_i]\!]^{\rm c}_{C_i,\mathcal{B}_{C_i}}$$

  from which it follows that

  $$[\![C_i]\!]^{\rm c}_{\mathcal{A}} \|_{\mathcal{S}(C_i,H;\mathcal{A})} [\![H]\!]^{\rm c}_{C_i,\mathcal{B}_{C_i}} \approx_{\mathcal{P}} [\![C_i]\!]^{\rm c}_{\mathcal{A}}$$

  Since $\approx_{\mathcal{P}}$ is a congruence with respect to the parallel composition operator,

  $$[\![C]\!]^{\rm c}_{\mathcal{A}'} \|_{\mathcal{S}(C,H;\mathcal{A}')} [\![H]\!]^{\rm c}_{C,\mathcal{B}_C} \approx_{\mathcal{P}} [\![C]\!]^{\rm c}_{\mathcal{A}'}$$

  because we hide interactions that are not attached to H (only C_i can be attached to H otherwise $\mathcal{CB}$ would not be a cyclic border), from which it follows that

  $$[\![C]\!]^{\rm c}_{C,\mathcal{B}_C} \|_{\mathcal{S}(C,H;C,\mathcal{B}_C)} [\![H]\!]^{\rm c}_{C,\mathcal{B}_C} \approx_{\mathcal{P}} [\![C]\!]^{\rm c}_{C,\mathcal{B}_C}$$

  On H side, it can similarly be shown that

  $$[\![H]\!]^{\rm c}_{H,\mathcal{B}_H} \|_{\mathcal{S}(H,C;H,\mathcal{B}_H)} [\![C]\!]^{\rm c}_{H,\mathcal{B}_H} \approx_{\mathcal{P}} [\![H]\!]^{\rm c}_{H,\mathcal{B}_H}$$

  · If $\mathcal{A}'$ is cyclic, then (3) is preserved. In fact, let $\mathcal{CB}'_\sigma$ be the set of cyclic borders for $\mathcal{A}'$ obtained from $\mathcal{CB}_\sigma$ by replacing in each original cyclic border every occurrence of $C_1, \ldots, C_n$ with C. Every cyclic border in $\mathcal{CB}'_\sigma$ that does not include C has a corresponding topologically conformant cyclic border in $\mathcal{CB}_\sigma$. On the other hand, if we take in $\mathcal{CB}'_\sigma$ a cyclic border formed by the AEIs $H_1, \ldots, H_l, C$, then $\mathcal{CB}_\sigma$ contains a cyclic border formed by the AEIs $H_1, \ldots, H_l, C_i$, where $C_i \in \mathcal{F}_{C_1,\ldots,C_n}$, because of Lemma 4.12. By virtue of (3).II

  $$[\![C_i]\!]^{\rm c}_{\mathcal{A}} \approx_{\mathcal{P}} [\![H_1, \ldots, H_l, C_i]\!]^{\rm c}_{\mathcal{A}} / (Act - \mathcal{S}(C_i;\mathcal{A}))$$

  Since $\approx_{\mathcal{P}}$ is a congruence with respect to the parallel composition operator,

$$[\![C]\!]^{\mathrm{c}}_{\mathcal{A}'} \approx_{\mathcal{P}} [\![H_1, \ldots, H_l, C]\!]^{\mathrm{c}}_{\mathcal{A}'}/(Act - \mathcal{S}(C; \mathcal{A}'))$$

because we hide interactions that do not occur in C. As a consequence, if $\mathcal{F}_{H_1,\ldots,H_l,C} = \emptyset$ then (3).I is preserved. On the other hand, if $C \in \mathcal{F}_{H_1,\ldots,H_l,C}$, then C preserves (3).II.

Similarly, for each $H_j \in \mathcal{F}_{H_1,\ldots,H_l,C} - \{C\}$, by (3).II applied to $H_1, \ldots, H_l, C_i$ we have

$$[\![H_j]\!]^{\mathrm{c}}_{\mathcal{A}} \approx_{\mathcal{P}} [\![H_1, \ldots, H_l, C_i]\!]^{\mathrm{c}}_{\mathcal{A}}/(Act - \mathcal{S}(H_j; \mathcal{A}))$$

From (3).II applied to $C_1, \ldots, C_n$ it follows

$$[\![C_i]\!]^{\mathrm{c}}_{\mathcal{A}} \approx_{\mathcal{P}} [\![C_1, \ldots, C_n]\!]^{\mathrm{c}}_{\mathcal{A}}/(Act - \mathcal{S}(C_i; \mathcal{A}))$$

Since $\approx_{\mathcal{P}}$ is a congruence with respect to the parallel composition operator, we have that

$$[\![H_j]\!]^{\mathrm{c}}_{\mathcal{A}'} \approx_{\mathcal{P}} [\![H_1, \ldots, H_l, C]\!]^{\mathrm{c}}_{\mathcal{A}'}/(Act - \mathcal{S}(H_j; \mathcal{A}'))$$

because we hide interactions that do not occur in H_j.

· The abstract enriched flow graph of $\mathcal{A}'$ has at most m cycles.

Then by the induction hypothesis it follows that $[\![\mathcal{A}']\!]^{\mathrm{c}}_{\mathrm{bbv}}$ satisfies $\mathcal{P}$. Since

$$[\![\mathcal{A}']\!]^{\mathrm{c}}_{\mathrm{bbv}} \approx_{\mathcal{P}} [\![\mathcal{A}]\!]^{\mathrm{c}}_{\mathrm{bbv}} / \Big(\bigcup_{C_i \notin \mathcal{F}_{C_1,\ldots,C_n}} \mathcal{S}(C_i; \mathcal{A}) - \bigcup_{C_i \in \mathcal{F}_{C_1,\ldots,C_n}} \mathcal{S}(C_i; \mathcal{A}) \Big)$$

we derive that $[\![\mathcal{A}]\!]^{\mathrm{c}}_{\mathrm{bbv}}$ satisfies $\mathcal{P}$, because $\approx_{\mathcal{P}}$ preserves $\mathcal{P}$ and $\mathcal{P}$ does not contain any free use of the negation. ■

It is worth pointing out that a violation of one of the three conditions of the theorem above does not necessarily imply a violation of $\mathcal{P}$ in $\mathcal{A}$, but reveals the possible presence of some kind of $\mathcal{P}$-related mismatch in a specific portion of the topology of $\mathcal{A}$. In this case, component-oriented diagnostic information can be derived as explained in Sect. 4.2.

As a simple example of application of Thm. 4.14 with $\mathcal{P}$ being deadlock freedom, we derive that the architectural type `Pipe_Filter` defined in Sect. 2 is deadlock free, because each of its AEIs is deadlock free and $\mathcal{P}$-compatible (using $\approx_{\mathrm{B}}$) with every AEI attached to it.

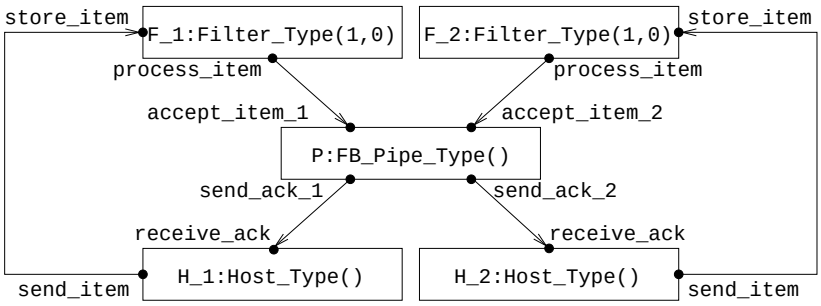

Fig. 7. Graphical description of `FB_Pipe_Filter`

As a more complicated example of application of Thm. 4.14, let us consider the architectural type `FB_Pipe_Filter` depicted in Fig. 7. This system is composed of two hosts, two filters of capacity one, and a pipe which is expected to feed

the items back to the hosts. Each host generates an item which is passed to the related filter. Every two generated items, the host waits for an ack from the pipe. Each filter processes the incoming item and sends it to the pipe. Upon sending an ack to a host, the pipe can accept two items generated by the other host and send back the related ack. Each new AET is defined below:

```
ELEM_TYPE Host_Type(void)

  BEHAVIOR
    Host(void; void) =
      send_item . send_item . receive_ack . Host()

  INPUT_INTERACTIONS  UNI receive_ack
  OUTPUT_INTERACTIONS UNI send_item

ELEM_TYPE FB_Pipe_Type(void)

  BEHAVIOR
    FB_Pipe(void; void) =
      choice {
        send_ack_1 . accept_item_2 .
          accept_item_2 . send_ack_2 . FB_Pipe(),
        send_ack_2 . accept_item_1 .
          accept_item_1 . send_ack_1 . FB_Pipe()
      }

  INPUT_INTERACTIONS  UNI accept_item_1; accept_item_2
  OUTPUT_INTERACTIONS UNI send_ack_1; send_ack_2
```

Suppose that the property $\mathcal{P}$ we are interested in is again deadlock freedom. The system deadlocks, because the pipe waits for sending an ack which none of the two hosts can receive, the filters cannot send to the pipe any of the items generated by the hosts, and each host is blocked just after sending the first item to the related filter. From the topology standpoint, each host forms a cycle with its dedicated filter and the pipe. Such cycles are not disjoint, as they all share the pipe, and applying the $\mathcal{P}$-interoperability check to any component but the pipe is not enough to detect deadlock. Formally, it can be verified along every single cycle $\mathtt{i}$ that $\mathtt{H_i}$ (resp. $\mathtt{F_i}$) $\mathcal{P}$-interoperates with $\mathtt{F_i}$ (resp. $\mathtt{H_i}$) and $\mathtt{P}$. As can easily be seen, the point is that in all cases we abstract away from the local interactions of $\mathtt{P}$ with the other host, which is not in the considered cycle. Therefore, we cannot take into account the influence of such a host upon the overall behavior of the cycle. To achieve that, following Thm. 4.14 and observed that conditions (1) and (2) are trivially satisfied, we consider e.g. the cyclic borders $\mathcal{CB}_{\mathtt{H_1}} = \{\mathtt{H_1}, \mathtt{F_1}, \mathtt{P}\}$ and $\mathcal{CB}_{\mathtt{H_2}} = \{\mathtt{H_2}, \mathtt{F_2}, \mathtt{P}\}$ obtained by applying a total cycle covering strategy that does not pick $\mathtt{P}$. It

can be verified that P, which represents the frontier for both cyclic borders, $\mathcal{P}$-interoperates neither with $\mathtt{H}_1$ and $\mathtt{F}_1$, nor with $\mathtt{H}_2$ and $\mathtt{F}_2$, which reveals a potential mismatch that, as we have seen before, actually causes a deadlock.

4.4 Behaviorally Conformant Architectural Invocations

The validity of an arbitrary property $\mathcal{P}$ for an architectural type easily scales to its architectural invocations in which only actual behavioral parameters are passed, which conform to the corresponding formal parameters. The only constraint is that the names of the local interactions occurring in $\mathcal{P}$ are preserved.

Theorem 4.15 Let $\mathcal{A}$ be an architectural type such that $[\![\mathcal{A}]\!]^{\mathrm{c}}_{\mathrm{bbv}}$ satisfies $\mathcal{P}$ and let $\mathcal{A}'$ be a behaviorally conformant architectural invocation of $\mathcal{A}$ preserving the names of the local interactions occurring in $\mathcal{P}$. Then $[\![\mathcal{A}']\!]^{\mathrm{c}}_{\mathrm{bbv}}$ satisfies $\mathcal{P}$.

Proof Due to behavioral conformity, $[\![\mathcal{A}']\!]^{\mathrm{c}}_{\mathrm{bbv}}$ and $[\![\mathcal{A}]\!]^{\mathrm{c}}_{\mathrm{bbv}}$ are weakly bisimulation equivalent up to an injective relabeling function that matches their local interactions. Since the names of the local interactions occurring in $\mathcal{P}$ are preserved and $\approx_{\mathrm{B}}$ implies $\approx_{\mathcal{P}}$, $[\![\mathcal{A}']\!]^{\mathrm{c}}_{\mathrm{bbv}}$ and $[\![\mathcal{A}]\!]^{\mathrm{c}}_{\mathrm{bbv}}$ are equivalent according to $\approx_{\mathcal{P}}$ up to an injective relabeling function that matches their local interactions. Since $[\![\mathcal{A}]\!]^{\mathrm{c}}_{\mathrm{bbv}}$ satisfies $\mathcal{P}$ and $\approx_{\mathcal{P}}$ preserves $\mathcal{P}$, then $[\![\mathcal{A}']\!]^{\mathrm{c}}_{\mathrm{bbv}}$ satisfies $\mathcal{P}$ as well. ■

As an example of application of Thm. 4.15 with $\mathcal{P}$ being deadlock freedom, we immediately derive that every behaviorally conformant invocation (like the one presented in Sect. 3.2) of the architectural type `Pipe_Filter` defined in Sect. 2 is deadlock free.

4.5 Generalization to Exogenous Extensions

We now consider the scalability of the validity of $\mathcal{P}$ from an architectural type $\mathcal{A}$, which satisfies the three conditions of Thm. 4.14 and possesses some architectural interactions, to one of its exogenous extensions. In this case there are two issues to be taken into account. The first issue is that all the architectural interactions at which the exogenous extension takes place become local interactions, which must satisfy the $\mathcal{P}$-compatibility and $\mathcal{P}$-interoperability checks whenever necessary. To this purpose, the AEIs of $\mathcal{A}$ containing the architectural interactions at which the exogenous extension takes place must undergo to an extended version of the $\mathcal{P}$-compatibility and $\mathcal{P}$-interoperability checks, in which such architectural interactions are left visible as well. The second

issue is that the exogenous extension may generate kinds of cycles that are not present in the topology of $\mathcal{A}$, in which case we cannot derive the validity of $\mathcal{P}$ for the exogenous extension based on the three conditions of Thm. 4.14 satisfied by $\mathcal{A}$. As an example of generation of such new kinds of cycles, consider a variant of the architectural type `Pipe_Filter` of Sect. 2 with several upstream filters, each of which has an architectural interaction and is attached to the `accept_item` interaction of the pipe that is now an or-interaction. Every exogenous extension of this topology results in several instances of a new kind of cycle, each involving two pipes and two of the filters in between.

We now define the open versions of the notions of interacting semantics, $\mathcal{P}$-compatibility, $\mathcal{P}$-interoperability, and frontier of a set of AEIs, in which the architectural interactions are left visible.

Definition 4.16 Let $\mathcal{A}$ be an architectural type and let $C_1, \ldots, C_n$ be some of its AEIs. For all $1 \leq i \leq n$, the open interacting semantics of C_i w.r.t. $C_1, \ldots, C_n$ is defined by

$$[\![C_i]\!]^{\mathrm{o}}_{C_1,\ldots,C_n} = [\![C_i]\!] \,/\, (Act - (\mathcal{LI}_{C_i;C_1,\ldots,C_n} \cup \mathcal{AI}_{C_i}))\,[\varphi_{C_i;C_1,\ldots,C_n}]$$ ■

Definition 4.17 Let $\mathcal{A}$ be an architectural type, let $C_1, \ldots, C_n$ be some of its AEIs, and let $C'_1, \ldots, C'_{n'}$ be some of such AEIs. The open interacting semantics of $C'_1, \ldots, C'_{n'}$ w.r.t. $C_1, \ldots, C_n$ is defined by

$$\begin{aligned}[\![C'_1, \ldots, C'_{n'}]\!]^{\mathrm{o}}_{C_1,\ldots,C_n} = {} & [\![C'_1]\!]^{\mathrm{o}}_{C_1,\ldots,C_n} \,\|_{\mathcal{S}(C'_1,C'_2;C'_1,\ldots,C'_{n'})} \\ & [\![C'_2]\!]^{\mathrm{o}}_{C_1,\ldots,C_n} \,\|_{\mathcal{S}(C'_1,C'_3;C'_1,\ldots,C'_{n'}) \cup \mathcal{S}(C'_2,C'_3;C'_1,\ldots,C'_{n'})} \cdots \\ & \cdots \,\|_{\cup_{i=1}^{n'-1} \mathcal{S}(C'_i,C'_{n'};C'_1,\ldots,C'_{n'})} [\![C'_{n'}]\!]^{\mathrm{o}}_{C_1,\ldots,C_n}\end{aligned}$$ ■

Definition 4.18 Let $\mathcal{A}$ be an architectural type and let $C_1, \ldots, C_n$ be all of its AEIs. The open semantics of $\mathcal{A}$ before the behavioral variations is defined by

$$[\![\mathcal{A}]\!]^{\mathrm{o}}_{\mathrm{bbv}} = [\![C_1, \ldots, C_n]\!]^{\mathrm{o}}_{C_1,\ldots,C_n}$$ ■

Definition 4.19 Let $\mathcal{A}$ be an architectural type, let K be one of its AEIs, and let $\mathcal{B}_K = \{C_1, \ldots, C_n\}$. We say that K is $\mathcal{P}^{\mathrm{o}}$-compatible with C_i iff

$$[\![K]\!]^{\mathrm{o}}_{K,\mathcal{B}_K} \,\|_{\mathcal{S}(K,C_i;K,\mathcal{B}_K)}\, [\![C_i]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \approx_{\mathcal{P}} [\![K]\!]^{\mathrm{o}}_{K,\mathcal{B}_K}$$ ■

Definition 4.20 Let $\mathcal{A}$ be an architectural type and let $C_1, \ldots, C_n$ be some of its AEIs forming a cycle in the abstract enriched flow graph of $\mathcal{A}$. We say that C_i $\mathcal{P}^{\mathrm{o}}$-interoperates with the other AEIs in the cycle iff

$$[\![C_1, \ldots, C_n]\!]^{\mathrm{o}}_{\mathcal{A}} \,/\, (Act - (\mathcal{S}(C_i;\mathcal{A}) \cup \mathcal{AI}_{C_i})) \approx_{\mathcal{P}} [\![C_i]\!]^{\mathrm{o}}_{\mathcal{A}}$$ ■

Observe that $\mathcal{P}^{\mathrm{o}}$-compatibility and $\mathcal{P}^{\mathrm{o}}$-interoperability imply $\mathcal{P}$-compatibility and $\mathcal{P}$-interoperability, respectively, because $\approx_{\mathcal{P}}$ is a congruence with respect to the hiding operator.

Definition 4.21 Let $\mathcal{A}$ be an architectural type and let $C_1, \ldots, C_n$ be some of its AEIs. The open frontier of $C_1, \ldots, C_n$ is defined by

$$\mathcal{F}^{o}_{C_1,\ldots,C_n} = \{C_i \in \{C_1, \ldots, C_n\} \mid \mathcal{AI}_{C_i} \neq \emptyset \vee \mathcal{LI}_{C_i;C_1,\ldots,C_n} \neq \mathcal{LI}_{C_i}\}$$ ■

We then introduce the notion of extensibility of a total cycle covering strategy to an exogenous extension. In the following, we denote by $\mathcal{CB}^{\mathcal{A}}_{\sigma}$ the set of cyclic borders generated by the cycle covering strategy σ applied to the architectural type $\mathcal{A}$, and by $\mathcal{CB}^{\mathcal{A}}_{K}$ the cyclic border of an AEI K of $\mathcal{A}$.

Definition 4.22 Let $\mathcal{A}$ be an architectural type, let σ be a total cycle covering strategy for $\mathcal{A}$, and let $\mathcal{A}'$ be an exogenous extension of $\mathcal{A}$. An exogenous extension of σ to $\mathcal{A}'$ is defined by the following algorithm:

(1) All the AEIs in the abstract enriched flow graph of $\mathcal{A}'$ are initially unmarked.
(2) For each $\mathcal{CB}^{\mathcal{A}}_{K} \in \mathcal{CB}^{\mathcal{A}}_{\sigma}$, pick out K and mark all the AEIs in $\mathcal{CB}^{\mathcal{A}'}_{K}$.
(3) While there is an unmarked additional AEI C in the cycles of the abstract enriched flow graph of $\mathcal{A}'$ such that there exists $\mathcal{CB}^{\mathcal{A}}_{C'} \in \mathcal{CB}^{\mathcal{A}}_{\sigma}$ with $C' = corr(C)$ and $\mathcal{CB}^{\mathcal{A}'}_{C}$ topologically conforming to $\mathcal{CB}^{\mathcal{A}}_{C'}$:
   (a) Pick out C.
   (b) Mark all the AEIs in $\mathcal{CB}^{\mathcal{A}'}_{C}$.

We say that $\mathcal{A}'$ is exo-coverable by σ iff all the AEIs in the cycles of the abstract enriched flow graph of $\mathcal{A}'$ are marked, the exogenous extension of σ is total, and for each AEI K in $\mathcal{A}$ such that $\mathcal{CB}^{\mathcal{A}}_{K} \in \mathcal{CB}^{\mathcal{A}}_{\sigma}$ it holds $\mathcal{CB}^{\mathcal{A}}_{K} = \mathcal{CB}^{\mathcal{A}'}_{K}$. In general, we say that $\mathcal{A}'$ is exo-coverable iff there exists a cycle covering strategy σ for $\mathcal{A}$ such that $\mathcal{A}'$ is exo-coverable by σ. ■

As a consequence of the previous definition, we observe that if $\mathcal{A}'$ is exo-coverable by σ then each cyclic border generated by the exogenous extension of σ to $\mathcal{A}'$ topologically conforms to a cyclic border generated by σ to $\mathcal{A}$. Therefore, if $\mathcal{A}$ is acyclic, then each cyclic exogenous extension $\mathcal{A}'$ of $\mathcal{A}$ cannot be exo-coverable. Moreover, if $\mathcal{A}$ has an arbitrary topology, then no exo-coverable exogenous extension of $\mathcal{A}$ can be a cyclic border with empty frontier.

Theorem 4.23 Let $\mathcal{A}$ be an architectural type with an arbitrary topology and at least one architectural interaction and let $\mathcal{A}'$ be an exogenous extension of $\mathcal{A}$. Suppose that $\mathcal{A}$ satisfies the three conditions of Thm. 4.14, with σ being the total cycle covering strategy of condition (3). Suppose that the three following additional conditions hold:

(4_{exo}) $\mathcal{A}'$ is exo-coverable by σ.
(5_{exo}) For every AEI K of $\mathcal{A}$ of the same type as an AEI having architectural interactions at which the exogenous extension takes place, K is $\mathcal{P}^{o}$-compatible with each AEI in $\{C \in \mathcal{B}_K \mid C \notin \mathcal{CB}^{\mathcal{A}}_{K}\}$.

(6_{exo}) If $\mathcal{A}$ is cyclic, then for every AEI C_i in the open frontier of a cyclic border $\{C_1, \ldots, C_n\}$ in $\mathcal{CB}^{\mathcal{A}}_{\sigma}$ that is of the same type of an AEI having architectural interactions at which the exogenous extension takes place, C_i $\mathcal{P}^{\text{o}}$-interoperates with $C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n$.

Then $[\![\mathcal{A}']\!]^{\text{c}}_{\text{bbv}}$ satisfies $\mathcal{P}$.

Proof We show that $\mathcal{A}'$ satisfies the three conditions of Thm. 4.14, from which the result follows.

- $\mathcal{A}'$ satisfies (1) because, by definition of exogenous extension, no new AET can be introduced with respect to $\mathcal{A}$.
- $\mathcal{A}'$ satisfies (2) by virtue of (5_{exo}) and by definition of exogenous extension. Consider an AEI K of $\mathcal{A}'$ and an AEI C of $\mathcal{A}'$ attached to it but not in $\mathcal{CB}^{\mathcal{A}'}_{K}$. If both AEIs are in $\mathcal{A}$, then K is $\mathcal{P}$-compatible with C by virtue of condition (2) of Thm. 4.14 applied to $\mathcal{A}$ in the case K is not an AEI having architectural interactions at which the exogenous extension takes place, or by virtue of (5_{exo}) otherwise. If K is in $\mathcal{A}$ and C is an additional AEI, in $\mathcal{A}$ there exists an attachment between an AEI K' and $corr(C)$, such that K' is of the same type as K and $corr(C) \notin \mathcal{CB}^{\mathcal{A}}_{K'}$. Then, by virtue of ($5_{\text{exo}}$), K' is $\mathcal{P}^{\text{o}}$-compatible with $corr(C)$, hence K is $\mathcal{P}$-compatible with C. We can argue similarly if K is an additional AEI and C is in $\mathcal{A}$. Finally, if both K and C are additional AEIs, from the definition of exogenous extension we derive that in $\mathcal{A}$ there exist two attached AEIs $corr(K)$ and $corr(C)$ such that, by virtue of condition (2) of Thm. 4.14 applied to $\mathcal{A}$, $corr(K)$ is $\mathcal{P}$-compatible with $corr(C)$. As a consequence, K is $\mathcal{P}$-compatible with C.
- If $\mathcal{A}'$ is cyclic, then $\mathcal{A}'$ satisfies (3). We first observe that $\mathcal{A}'$ trivially satisfies (3).I because, by virtue of (4_{exo}), the exogenous extension of σ to $\mathcal{A}'$ cannot generate a single cyclic border with empty frontier.
  Suppose now that $\mathcal{CB}^{\mathcal{A}'}_{K}$ is a cyclic border generated by the exogenous extension of σ, which is total since so is σ. By virtue of (4_{exo}), we distinguish between two possible cases:
  · If K is in $\mathcal{A}$, then, by virtue of (4_{exo}), $\mathcal{CB}^{\mathcal{A}'}_{K} = \mathcal{CB}^{\mathcal{A}}_{K}$ with $\mathcal{CB}^{\mathcal{A}}_{K} \in \mathcal{CB}^{\mathcal{A}}_{\sigma}$ and each $C_i \in \mathcal{F}_{\mathcal{CB}^{\mathcal{A}'}_{K}}$ belonging to $\mathcal{F}^{\text{o}}_{\mathcal{CB}^{\mathcal{A}}_{K}}$ as well. Then, by virtue of (6_{exo}) or condition (3).II of Thm. 4.14 applied to $\mathcal{A}$, C_i $\mathcal{P}$-interoperates with the other AEIs of $\mathcal{CB}^{\mathcal{A}'}_{K}$, hence $\mathcal{CB}^{\mathcal{A}'}_{K}$ satisfies (3).II.
  · If K is an additional AEI, then, by virtue of (4_{exo}), $\mathcal{CB}^{\mathcal{A}'}_{K}$ topologically conforms to $\mathcal{CB}^{\mathcal{A}}_{corr(K)} \in \mathcal{CB}^{\mathcal{A}}_{\sigma}$. Since $\mathcal{CB}^{\mathcal{A}}_{corr(K)}$ satisfies (3).II by hypothesis, by means of an argument similar to that applied above it follows that $\mathcal{CB}^{\mathcal{A}'}_{K}$ satisfies (3).II as well. ■

As an example of application of Thm. 4.23 with $\mathcal{P}$ being deadlock freedom, we immediately derive that every exogenous extension (like the one presented in Sect. 3.3) of the architectural type `Pipe_Filter` defined in Sect. 2 with an

arbitrary number of additional pipes and filters is deadlock free.

4.6 Generalization to Endogenous Extensions

In this section we address the scalability of the validity of $\mathcal{P}$ from an architectural type $\mathcal{A}$, which satisfies the three conditions of Thm. 4.14, to one of its endogenous extensions. In this case there are two issues to be taken into account. The first issue is that, as observed in Sect. 3.4, the endogenous extension may introduce kinds of attachments that are not present in the topology of $\mathcal{A}$, in which case we cannot derive the validity of $\mathcal{P}$ for the endogenous extension based on the three conditions of Thm. 4.14 satisfied by $\mathcal{A}$. The second issue is that the endogenous extension may alterate the cyclic borders of $\mathcal{A}$. From the point of view of the scalability of the validity of $\mathcal{P}$, this is dealt with by admitting only certain modifications of the original cyclic borders, ruling out in particular new kinds of cycles that are not present in the topology of $\mathcal{A}$.

Before showing the scalability result, we introduce the notion of extensibility of a cycle covering strategy to an endogenous extension.

Definition 4.24 Let $\mathcal{A}$ be an architectural type, let σ be a total cycle covering strategy for $\mathcal{A}$, and let $\mathcal{A}'$ be an endogenous extension of $\mathcal{A}$ adding the AEIs $C_1, \ldots, C_m$. An endogenous extension of σ to $\mathcal{A}'$ is defined by the following algorithm:

(1) All the AEIs in the abstract enriched flow graph of $\mathcal{A}'$ are initially unmarked.
(2) For each $\mathcal{CB}_K^{\mathcal{A}} \in \mathcal{CB}_\sigma^{\mathcal{A}}$, pick out K and mark all the AEIs in $\mathcal{CB}_K^{\mathcal{A}'}$.
(3) While there is an unmarked additional AEI C in the cycles of the abstract enriched flow graph of $\mathcal{A}'$ such that there exists $\mathcal{CB}_{C'}^{\mathcal{A}} \in \mathcal{CB}_\sigma^{\mathcal{A}}$ with C' of the same type as C and $\mathcal{CB}_C^{\mathcal{A}'}$ topologically conforming to $\mathcal{CB}_{C'}^{\mathcal{A}}$:
   (a) Pick out C.
   (b) Mark all the AEIs in $\mathcal{CB}_C^{\mathcal{A}'}$.

We say that $\mathcal{A}'$ is endo-coverable by σ iff all the AEIs in the cycles of the abstract enriched flow graph of $\mathcal{A}'$ are marked, the endogenous extension of σ is total, and for each AEI K in $\mathcal{A}$ such that $\mathcal{CB}_K^{\mathcal{A}} \in \mathcal{CB}_\sigma^{\mathcal{A}}$ it holds $\mathcal{CB}_K^{\mathcal{A}'} - \{C_1, \ldots, C_m\} = \mathcal{CB}_K^{\mathcal{A}}$. In general, we say that $\mathcal{A}'$ is endo-coverable iff there exists a cycle covering strategy σ for $\mathcal{A}$ such that $\mathcal{A}'$ is endo-coverable by σ. ■

As a consequence of the previous definition, if $\mathcal{A}$ is acyclic, then each endo-coverable endogenous extension of $\mathcal{A}$ is acyclic.

Theorem 4.25 Let $\mathcal{A}$ be an architectural type with an arbitrary topology and let $\mathcal{A}'$ be an endogenous extension of $\mathcal{A}$ adding the AEIs $C_1, \ldots, C_m$. Suppose that $\mathcal{A}$ satisfies the three conditions of Thm. 4.14, with σ being the total cycle covering strategy of condition (3). Suppose that the three following additional conditions hold:

(4_{endo}) $\mathcal{A}'$ is endo-coverable by σ.
(5_{endo}) For every attachment in $\mathcal{A}'$ from an AEI K_1 to another AEI K_2, there exists an attachment in $\mathcal{A}$ from an AEI of the same type as K_1 to another AEI of the same type as K_2.
(6_{endo}) Let $\mathcal{LI}$ be the set of local interactions of $C_1, \ldots, C_m$ that are not attached to interactions of the AEIs of $\mathcal{A}$. For every $\mathcal{CB}^{\mathcal{A}}_K \in \mathcal{CB}^{\mathcal{A}}_\sigma$ it holds $[\![\mathcal{CB}^{\mathcal{A}'}_K]\!]^{\text{c}}_{\mathcal{CB}^{\mathcal{A}'}_K} \,/\, \mathcal{LI} \approx_{\mathcal{P}} [\![\mathcal{CB}^{\mathcal{A}}_K]\!]^{\text{c}}_{\mathcal{CB}^{\mathcal{A}}_K}$.

Then $[\![\mathcal{A}']\!]^{\text{c}}_{\text{bbv}}$ satisfies $\mathcal{P}$.

Proof We show that $\mathcal{A}'$ satisfies the three conditions of Thm. 4.14, from which the result follows.

- $\mathcal{A}'$ satisfies (1) because, by definition of endogenous extension, no new AET can be introduced with respect to $\mathcal{A}$.
- $\mathcal{A}'$ satisfies (2) by virtue of (5_{endo}) and by definition of endogenous extension. Consider an AEI K and an AEI C attached to it but not in $\mathcal{CB}^{\mathcal{A}'}_K$. If both AEIs are in $\mathcal{A}$, then K is $\mathcal{P}$-compatible with C by virtue of condition (2) of Thm. 4.14 applied to $\mathcal{A}$. If K (resp. C) is in $\mathcal{A}$ and C (resp. K) is in $\{C_1, \ldots, C_m\}$, then, by definition of endogenous extension, in $\mathcal{A}$ we have that K (resp. C) is attached to an AEI C' (resp. K') that is of the same type as C (resp. K). From this we derive that K is $\mathcal{P}$-compatible with C by virtue of condition (2) of Thm. 4.14 applied to $\mathcal{A}$. Finally, if both K and C are in $\{C_1, \ldots, C_m\}$, then, by virtue of (5_{endo}), in $\mathcal{A}$ there exist two attached AEIs K' and C' of the same type as K and C, respectively, such that K' is $\mathcal{P}$-compatible with C'. As a consequence, K is $\mathcal{P}$-compatible with C.
- If $\mathcal{A}'$ is cyclic, then $\mathcal{A}'$ satisfies (3). First, by virtue of (4_{endo}), the endogenous extension of σ is total.
  Suppose now that the endogenous extension of σ generates a single cyclic border $\mathcal{CB}^{\mathcal{A}'}_K$ with empty frontier. Then, by virtue of (4_{endo}), $\mathcal{CB}^{\mathcal{A}}_K \in \mathcal{CB}^{\mathcal{A}}_\sigma$ and $\mathcal{CB}^{\mathcal{A}'}_K - \{C_1, \ldots, C_m\} = \mathcal{CB}^{\mathcal{A}}_K$. Assume $\mathcal{CB}^{\mathcal{A}}_K = \{K, H, K_1, \ldots, K_n\}$ and let H be the AEI that, by virtue of Thm. 4.14 applied to $\mathcal{A}$, $\mathcal{P}$-interoperates with $K, K_1, \ldots, K_n$. Then, by virtue of (6_{endo}), H $\mathcal{P}$-interoperates with the other AEIs in $\mathcal{CB}^{\mathcal{A}'}_K$, which means that $\mathcal{A}'$ satisfies (3).I.
  Suppose now that $\mathcal{CB}^{\mathcal{A}'}_K$ is a cyclic border generated by the endogenous extension of σ. By virtue of (4_{endo}), we distinguish between two possible cases:
  · If $\mathcal{CB}^{\mathcal{A}'}_K$ is equal or topologically conforms to $\mathcal{CB}^{\mathcal{A}}_K \in \mathcal{CB}^{\mathcal{A}}_\sigma$, then $\mathcal{CB}^{\mathcal{A}'}_K$

satisfies (3).II by virtue of condition (3).II of Thm. 4.14 applied to $\mathcal{A}$.

- If $\mathcal{CB}_K^{\mathcal{A}'} - \{C_1, \ldots, C_m\} = \mathcal{CB}_K^{\mathcal{A}}$ with $\mathcal{CB}_K^{\mathcal{A}} \in \mathcal{CB}_\sigma^{\mathcal{A}}$, in order to derive (3).II we exploit (6_{endo}) and an argument similar to that applied for (3).I. In particular, for each $C \in \mathcal{F}_{\mathcal{CB}_K^{\mathcal{A}}}$, we have $C \in \mathcal{F}_{\mathcal{CB}_K^{\mathcal{A}'}}$. Hence, (3).II trivially holds. For each $C \in \mathcal{F}_{\mathcal{CB}_K^{\mathcal{A}'}}$ such that $C \in \{C_1, \ldots, C_m\}$, we have that C is attached to an AEI C' not in $\mathcal{CB}_K^{\mathcal{A}'}$ iff there exists $C'' \in \mathcal{F}_{\mathcal{CB}_K^{\mathcal{A}}}$ – of the same type as C – that is attached to an AEI C''' – of the same type as C' – not in $\mathcal{CB}_K^{\mathcal{A}'}$. Then, since C'' satisfies (3).II by virtue of condition (3).II of Thm. 4.14 applied to $\mathcal{A}$, we have that C satisfies (3).II as well. ■

As an example of application of Thm. 4.25 with $\mathcal{P}$ being deadlock freedom, let us consider the architectural type `Station_Ring` defined in Sect. 3.4. If we take the endogenous extension `Station_Ring(2)` of `Station_Ring(1)`, then Thm. 4.25 does not apply because of the violation of condition (5_{endo}) due to the introduction of a new kind of attachment (the one between two normal stations). Let us take instead any endogenous extension `Station_Ring(n)` of `Station_Ring(2)` with `n` > 2. Observed that every AEI of `Station_Ring(2)` is deadlock free and $\mathcal{P}$-interoperates with the other AEIs – thus `Station_Ring(2)` is deadlock free by virtue of Thm. 4.14 – since the three additional conditions of Thm. 4.25 are satisfied it follows that `Station_Ring(n)` is deadlock free as well.

4.7 Generalization to And/Or Extensions

Similarly to the exogenous and endogenous extensions, we guarantee the scalability of the validity of $\mathcal{P}$ from an architectural type $\mathcal{A}$, which satisfies the three conditions of Thm. 4.14, to one of its and/or extensions whenever no new cycles are added by the extension itself. Unlike the previous two cases of extensions, we do not need to introduce a concept of and/or extension for a cycle covering strategy, because the set of cyclic borders of the and/or extension of $\mathcal{A}$ coincides with the set of cyclic borders of $\mathcal{A}$.

Theorem 4.26 Let $\mathcal{A}$ be an architectural type with an arbitrary topology and let $\mathcal{A}'$ be an and/or extension of $\mathcal{A}$. Suppose that $\mathcal{A}$ satisfies the three conditions of Thm. 4.14, with σ being the total cycle covering strategy of condition (3). Suppose that the two following additional conditions hold:

(4_{andor}) Every extended or-interaction is enabled infinitely often.
(5_{andor}) No additional AEI belongs to a cycle of the abstract enriched flow graph of $\mathcal{A}'$.

Then $[\![\mathcal{A}']\!]^{\text{c}}_{\text{bbv}}$ satisfies $\mathcal{P}$.

Proof We show that $\mathcal{A}'$ satisfies the three conditions of Thm. 4.14, from which the result follows.

- $\mathcal{A}'$ satisfies (1) because, by definition of and/or extension, no new AET can be introduced with respect to $\mathcal{A}$.
- $\mathcal{A}'$ satisfies (2) by virtue of (4_{andor}) and by definition of and/or extension. Consider an AEI K and an AEI C attached to it but not in $\mathcal{CB}_K^{\mathcal{A}'}$. If both K and C are in $\mathcal{A}$ and are not attached through an and/or-interaction that is extended, then K is $\mathcal{P}$-compatible with C by virtue of condition (2) of Thm. 4.14 applied to $\mathcal{A}$. Now suppose that K is in $\mathcal{A}$ and has an and- or an or-interaction that is extended. If C is one of the AEIs attached to the considered interaction of K, then C is of the same type as an AEI C' of $\mathcal{A}$ that is attached to K. By virtue of condition (2) of Thm. 4.14 applied to $\mathcal{A}$, K is $\mathcal{P}$-compatible with C', from which we derive that K is $\mathcal{P}$-compatible with C. Suppose instead that C is in $\mathcal{A}$ and has an and-interaction that is extended. If K is one of the AEIs attached to the considered interaction of C, then K is of the same type as an AEI K' of $\mathcal{A}$ that is attached to C. Hence, by virtue of condition (2) of Thm. 4.14 applied to $\mathcal{A}$, K' is $\mathcal{P}$-compatible with C, from which it follows that K is $\mathcal{P}$-compatible with C. Finally, suppose that C is in $\mathcal{A}$ and has an or-interaction that is extended. If K is one of the AEIs attached to the considered interaction of C, then K is $\mathcal{P}$-compatible with C by virtue of an argument similar to the previous one together with (4_{andor}).
- $\mathcal{A}'$ satisfies (3) because, by virtue of (5_{andor}), the set of cyclic borders generated by σ for $\mathcal{A}'$ is the same as that generated by σ for $\mathcal{A}$. ■

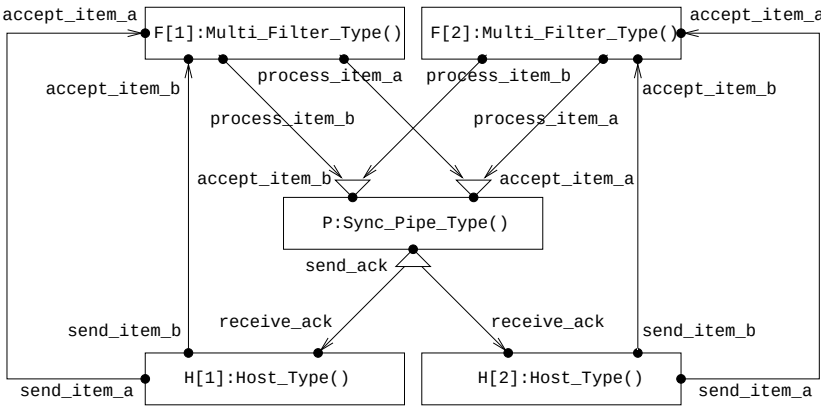

Fig. 8. Graphical description of `Sync_Pipe_Filter` with two hosts/filters

As an example of motivation for condition (5_{andor}) of Thm. 4.26, let us consider the architectural type `Sync_Pipe_Filter` depicted in Fig. 8. This system is composed of a certain number of hosts and filters of capacity one processing two different types of items – `a` and `b` – and a pipe that synchronizes the items processed by all the filters whenever they are of the same type, in which case the pipe acknowledges all the hosts. The AETs are defined below:

```
ELEM_TYPE Host_Type(void)

  BEHAVIOR
    Host(void; void) =
      choice {
        send_item_a . receive_ack . Host(),
        send_item_b . receive_ack . Host()
      }

  INPUT_INTERACTIONS  UNI receive_ack
  OUTPUT_INTERACTIONS UNI send_item_a; send_item_b

ELEM_TYPE Multi_Filter_Type(void)

  BEHAVIOR
    Multi_Filter(void; void) =
      choice {
        accept_item_a . Multi_Filter_a(),
        accept_item_b . Multi_Filter_b(),
        fail . repair . Multi_Filter()
      };
    Multi_Filter_a(void; void) =
      choice {
        process_item_a . Multi_Filter(),
        fail . repair . Multi_Filter_a()
      };
    Multi_Filter_b(void; void) =
      choice {
        process_item_b . Multi_Filter(),
        fail . repair . Multi_Filter_b()
      }

  INPUT_INTERACTIONS  UNI accept_item_a; accept_item_b
  OUTPUT_INTERACTIONS UNI process_item_a; process_item_b

ELEM_TYPE Sync_Pipe_Type(void)

  BEHAVIOR
    Sync_Pipe(void; void) =
      choice {
        accept_item_a . send_ack . Sync_Pipe(),
        accept_item_b . send_ack . Sync_Pipe()
      }
```

```
INPUT_INTERACTIONS  AND accept_item_a; accept_item_b
OUTPUT_INTERACTIONS AND send_ack
```

Suppose that the property $\mathcal{P}$ we are interested in is deadlock freedom. From Thm. 4.14 we obtain that `Sync_Pipe_Filter(1)` is deadlock free, because `H[1]` is deadlock free and $\mathcal{P}$-interoperates with `F[1]` and `P`. Let us now consider its and/or extension `Sync_Pipe_Filter(2)` depicted in Fig. 8. Each host forms a cycle with its dedicated filter and the pipe. Such cycles are not disjoint, as they all share the pipe. Consider the scenario where `F[1]` processes an item of type `a`, while `F[2]` processes an item of type `b`. The cycle composed of `H[1]`, `F[1]`, and `P` deadlocks since `H[1]` waits for an acknowledgement from `P`, `F[1]` waits for delivering the item of type `a` to `P`, and `P` waits for an item of the same type from `F[2]`. On the other hand, `F[2]` is blocked since it is trying to send out an item of type `b` to `P` and, as a consequence, `H[2]` is blocked until the reception of an acknowledgement that `P` cannot send. As can easily be seen, the point is that `P` is involved in the additional cycle that is introduced by the and/or extension, and the influence of such a cycle upon the overall behavior of the system cannot be inferred a priori. From the viewpoint of the and/or extension, what happens is that all the additional AEIs are involved in a new cycle.

5 Conclusion

In this paper we have tackled the usability problem for process algebra. On the modeling side, we have proposed a set of guidelines to lift process algebra to a fully fledged ADL for the hierarchical design of parameterized system families, in a way that hides the process algebraic technicalities. Most of these guidelines have been incorporated in Æmilia, the process algebraic ADL implemented in the software tool TwoTowers [6]. On the verification side, we have proposed a technique based on equivalence checking for the detection of architectural mismatches and the provision of component-oriented diagnostic information for process algebraic architectural descriptions of system families. This technique – which will be implemented in TwoTowers – extends previous results in terms of generality of the considered mismatches, generality of the considered topologies, and scalability to system families.

Although the focus of this paper is the usability of process algebra, it is worth noting that this study has some general implications on the architectural design process. Compared to the informal box-and-line diagrams that are commonly used in practice, adopting an architecturally enhanced process algebra, together with the related component-oriented technique for mismatch detection, strengthens the architectural design process itself in terms of modeling

accuracy and property analyzability.

As further steps towards the solution of the usability problem, there are two main directions that we would like to investigate. The first one is related to dynamic architectures, which are typical of nowadays mobile communications and self-organizing systems. In this respect, we would like to understand whether our approach can be extended to deal with systems in which the number of components and connectors and the links among them can vary at run time.

The second direction is related to embedding our approach in the system development cycle. On the upstream side, this amounts to synthesize process algebraic architectural descriptions from the user requirements expressed in some notation, like e.g. UML, which is widely used in practice but does not fully support analyzability. On the downstream side, instead, this amounts to automatically generate code that is guaranteed to satisfy certain properties as formally proved at the architectural level, together with a set of representative tests to be used when deploying the system on a specific platform. Some preliminary work on code generation can be found in [7].

Acknowledgements

We wish to thank the anonymous referees for their useful comments and suggestions.

References

[1] A. Aldini and M. Bernardo, *"A General Deadlock Detection Approach for Software Architectures"*, in Proc. of the *12th Int. Formal Methods Europe Symp. (FME 2003)*, LNCS 2805:658-677, Pisa (Italy), 2003.

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

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

[4] J.C.M. Baeten and W.P. Weijland, *"Process Algebra"*, Cambridge University Press, 1990.

[5] J.A. Bergstra, A. Ponse, and S.A. Smolka (eds.), *"Handbook of Process Algebra"*, Elsevier, 2001.

[6] M. Bernardo, *"TwoTowers 5.0 User Manual"*, `http://www.sti.uniurb.it/bernardo/twotowers/`, 2004.

[7] M. Bernardo and E. Bontà, *"Generating Well-Synchronized Multithreaded Programs from Software Architecture Descriptions"*, in Proc. of the *4th Working IEEE/IFIP Conf. on Software Architecture (WICSA 2004)*, IEEE-CS Press, pp. 167-176, Oslo (Norway), 2004.

[8] M. Bernardo, P. Ciancarini, and L. Donatiello, *"Architecting Families of Software Systems with Process Algebras"*, in ACM Trans. on Software Engineering and Methodology 11:386-426, 2002.

[9] M. Bernardo and F. Franzè, *"Architectural Types Revisited: Extensible And/Or Connections"*, in Proc. of the *5th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2002)*, LNCS 2306:113-128, Grenoble (France), 2002.

[10] M. Bernardo and F. Franzè, *"Exogenous and Endogenous Extensions of Architectural Types"*, in Proc. of the *5th Int. Conf. on Coordination Models and Languages (COORDINATION 2002)*, LNCS 2315:40-55, York (UK), 2002.

[11] T. Bolognesi and E. Brinksma, *"Introduction to the ISO Specification Language LOTOS"*, in Computer Networks and ISDN Systems 14:25-59, 1987.

[12] C. Canal, E. Pimentel, and J.M. Troya, *"Compatibility and Inheritance in Software Architectures"*, in Science of Computer Programming 41:105-138, 2001.

[13] H. Garavel and M. Sighireanu, *"A Graphical Parallel Composition Operator for Process Algebras"*, in Proc. of the *IFIP Joint Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV 1999)*, Kluwer, pp. 185-202, Beijing (China), 1999.

[14] G. Gössler and J. Sifakis, *"Composition for Component-Based Modeling"*, in Proc. of the *1st Int. Symp. on Formal Methods for Components and Objects (FMCO 2002)*, LNCS 2852:443-466, Leiden (The Netherlands), 2003.

[15] C.A.R. Hoare, *"Communicating Sequential Processes"*, Prentice Hall, 1985.

[16] P. Inverardi and S. Uchitel, *"Proving Deadlock Freedom in Component-Based Programming"*, in Proc. of the *4th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2001)*, LNCS 2029:60-75, Genova (Italy), 2001.

[17] P. Inverardi, A.L. Wolf, and D. Yankelevich, *"Static Checking of System Behaviors Using Derived Component Assumptions"*, in ACM Trans. on Software Engineering and Methodology 9:239-272, 2000.

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

[19] J. Magee and J. Kramer, *"Concurrency: State Models & Java Programs"*, Wiley, 1999.

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

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

[22] M. Shaw and D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline"*, Prentice Hall, 1996.