A General Approach to Deadlock Freedom Verification for Software Architectures

Alessandro Aldini and Marco Bernardo

Università di Urbino "Carlo Bo"
Istituto di Scienze e Tecnologie dell'Informazione
Piazza della Repubblica 13, 61029 Urbino, Italy
{aldini, bernardo}@sti.uniurb.it

Abstract. When building complex software systems, the designer is faced with the problem of detecting mismatches arising from the activity of assembling components. The adoption of formal methods becomes unavoidable in order to support a precise identification of such mismatches in the early design stages. As far as deadlock freedom is concerned, some techniques appeared in the literature, which apply to formal specifications of software architectures under some constraints. In this paper we develop a novel technique for deadlock freedom verification that can be applied to arbitrary software architectures, thus overcoming the limitations of the previous techniques.

Keywords: software architecture, deadlock, process algebra.

1 Introduction

The software architecture level of design enables us to cope with the increasing size and complexity of nowadays software systems during the early stages of their development [10, 11]. To achieve this, the focus is turned from algorithms and data structures to the overall architecture of a software system, where the architecture is meant to be a collection of computational components together with a description of their interactions. As software architecture emerges as a discipline within software engineering, it becomes increasingly important to support architectural development with languages and tools. It is widely recognized that suitable architectural description languages (ADLs) should be devised to formalize software architectures instead of using informal box-and-line diagrams, and companion tools should be implemented to support the automatic analysis of architectural properties in order to allow the designer to make principled choices. Among the formal method based ADLs appeared in the literature, we mention those relying on process algebras [2, 8, 3], Z [1], and the CHAM [6].

Complex software systems are typically made out of numerous components whose behavior is individually well known. The main problem faced by a software designer is that of understanding whether the components fit together well. If the architecture of a software system is given a formal description, then adequate

techniques can hopefully be used to prove the well formedness of the system or to single out the components responsible for architectural mismatches. There are different kinds of architectural mismatches. A typical mismatch, which we address in this paper, is deadlock: starting from deadlock free components, the designer constructs a system that can deadlock. To adequately support the deadlock freedom verification at the architectural level of design, techniques must be developed that are scalable – because of the high number of components – and provide diagnostic information in case of mismatch – in order to know which part of the architecture must be modified.

In [2] a deadlock freedom verification technique has been developed, which exploits notions of equivalence defined for process algebra and considers single pairs of interactions of components communicating to each other. In [7] 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 [3] an even more general technique has been presented, which considers not only the interactions between pairs of components, but also the interactions within sets of components forming a ring. The last technique has been proved to scale to families of software architectures, called architectural types, that admit a controlled variability of the component internal behavior and of the architectural topology [4, 5].

The current limitation of the technique of [3] is that it addresses only specific topologies: acyclic topologies and ring topologies. More precisely, two deadlock related architectural checks have been defined. The first one, called architectural compatibility check, is concerned with architectural types whose topology is acyclic. For an acyclic architectural type, if we take a component K and we consider all the components $C_1, \ldots, C_n$ attached to it, we can observe that they form a star topology whose center is K, as the absence of cycles prevents any two components among $C_1, \ldots, C_n$ from communicating via a component different from K. It can easily be recognized that an acyclic architectural type is just a composition of star topologies. By means of a weak bisimulation equivalence [9] based condition to be locally verified on each pair of components in the star topology, the architectural compatibility check ensures the absence of deadlock within a star topology whose center K is deadlock free, and this check scales to the whole acyclic architectural type. The second check, called architectural interoperability check, deals with ring topologies. Also in this case, a weak bisimulation equivalence based condition is employed, which can be verified in a rather efficient way and guarantees the absence of deadlock within a ring of components in case that at least one of them is deadlock free.

In this paper we overcome the limitation of [3] by proposing a general and scalable deadlock freedom verification technique for architectural types with an arbitrary topology. From a conceptual viewpoint, the idea underlying the new technique is that an acyclic topology is a special topology to which every topology can be reduced. Given an arbitrary topology that is not acyclic, we reduce every cyclic portion of the topology satisfying the interoperability check into a single equivalent component, until we obtain an architectural type not satisfying the

check or we end up with an acyclic topology. From a practical viewpoint, the technique is implemented without actually having to reduce the topology. All we have to do is to apply a modified interoperability check, which is still based on the weak bisimulation equivalence, to some specific components of the topology.

This paper is organized as follows. In Sect. 2 we recall PADL, the process algebra based ADL of [3] that is used to formalize architectural types. In Sect. 3 we present our technique for detecting deadlock related architectural mismatches in arbitrary topologies. Finally, in Sect. 4 we report some concluding remarks.

2 Software Architecture Description

In this section we provide an overview of PADL, a process algebra based architectural description language for the representation of families of software systems, whose members share common component behaviors as well as common topologies. We start by recalling some notions about process algebra, then we present the syntax and the semantics for PADL. For more details, case studies, and comparisons with related work, the interested reader is referred to [3–5].

2.1 Process Algebra

The basic elements of any process algebra (see, e.g., [9]) are its actions, which represent activities carried out by the systems being modeled, and its operators – including a parallel composition operator – which are used to compose process algebraic descriptions.

The set of process terms of the process algebra PA that we consider in this paper is generated by the following syntax:

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

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

In the syntax above, $\underline{0}$ is the term that cannot execute any action. Term $a.E$ can execute action a and then behaves as term E. Term E/L behaves as term E with each executed action a turned into τ whenever $a \in L$. Term $E[\varphi]$ behaves as term E with each executed action a turned into $\varphi(a)$. Term $E_1 + E_2$ behaves as either term E_1 or term E_2 depending on whether an action of E_1 or an action of E_2 is executed. Term $E_1 \parallel_S E_2$ asynchronously executes actions of E_1 or E_2 not belonging to S and synchronously executes equal actions of E_1 and E_2 belonging to S. The action prefix operator "." and the alternative composition operator "+" are called dynamic operators, whereas the hiding operator "/", the relabeling operator "[]", and the parallel composition operator "$\parallel$" are called static operators. A term is called sequential if it is composed of dynamic operators only.

The semantics for PA is defined in the standard operational style by means of a set of axioms and inference rules, which formalize the meaning of each operator.

The result of the application of the operational semantic rules to a term E is a state transition graph $\mathcal{I}[\![E]\!]$, where states are in correspondence with process terms and transitions are labeled with actions. In order to get finitely branching state transition graphs, as usual we restrict ourselves to closed and guarded terms, i.e. we require that every constant has exactly one defining equation and every constant occurrence is within the scope of an action prefix operator.

Due to their algebraic nature, process description languages like PA naturally lend themselves to the definition of equivalences. The notion of equivalence that we consider in this paper is the weak bisimulation equivalence [9], denoted $\approx_{\mathrm{B}}$, which captures the ability of two terms to simulate each other behaviors up to τ actions. This equivalence has several useful properties that we shall exploit in the rest of the paper. First, $\approx_{\mathrm{B}}$ is able to abstract from unobservable details, as witnessed by the following equational laws:

$$\tau.E \approx_{\mathrm{B}} E$$
$$a.\tau.E \approx_{\mathrm{B}} a.E$$
$$E + \tau.E \approx_{\mathrm{B}} \tau.E$$
$$a.(E_1 + \tau.E_2) + a.E_2 \approx_{\mathrm{B}} a.(E_1 + \tau.E_2)$$

Second, $\approx_{\mathrm{B}}$ is a congruence with respect to the static operators: whenever $E_1 \approx_{\mathrm{B}} E_2$, then

$$E_1/L \approx_{\mathrm{B}} E_2/L$$
$$E_1[\varphi] \approx_{\mathrm{B}} E_2[\varphi]$$
$$E_1 \parallel_S E \approx_{\mathrm{B}} E_2 \parallel_S E$$

Finally, $\approx_{\mathrm{B}}$ preserves deadlock freedom, i.e. it never equates a term whose semantic model has a state from which no other state can be reached by executing an observable action – possibly preceded by τ actions – to a term whose semantic model is deadlock free, i.e. a term that has not such a state.

2.2 PADL Syntax

PADL is an architectural description language, equipped with both a textual notation and a graphical notation, that makes explicit the inherent component orientation of process algebra. A PADL description represents an architectural type. As shown in Table 1, each architectural type is defined as a function of its architectural element types (AETs) and its architectural topology. An AET is defined as a function of its behavior, specified either as a family of sequential PA terms or through an invocation of a previously defined architectural type, and its interactions, specified as a set of PA actions occurring in the behavior that act as interfaces for the AET. The architectural topology is specified through the declaration of a set of architectural element instances (AEIs) representing the system components, a set of architectural (as opposed to local) interactions given by some interactions of the AEIs that act as interfaces for the whole architectural type, and a set of directed architectural attachments among the interactions of the AEIs. Graphically, the AEIs are depicted as boxes, the local interactions are depicted as black circles, the architectural interactions are depicted as white squares, and the attachments are depicted as directed edges between pairs of attachments.

ARCHI_TYPE	⟨name⟩	
ARCHI_ELEM_TYPES	⟨architectural element types: behaviors and interactions⟩	
ARCHI_TOPOLOGY		
ARCHI_ELEM_INSTANCES	⟨architectural element instances⟩	
ARCHI_INTERACTIONS	⟨architectural interactions⟩	
ARCHI_ATTACHMENTS	⟨architectural attachments⟩	
END		

Table 1. Structure of a PADL textual description

Every interaction is declared to be an input interaction or an output interaction and the attachments must respect such a classification: every attachment must involve an output interaction and an input interaction of two different AEIs. In addition, every interaction is declared to be a uni-interaction, an and-interaction, or an or-interaction. As shown in Fig. 1, the only legal attachments are those between two uni-interactions, an and-interaction and a uni-interaction, and an or-interaction and a uni-interaction. 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. In the case of execution of an or-interaction, instead, it synchronizes with only one of the uni-interactions attached to it. An AEI can have different types of interactions (input/output, uni/and/or, local/architectural). Every local interaction must be involved in at least one attachment, while every architectural interaction must not be involved in any attachment. No isolated groups of AEIs are admitted in the architectural topology.

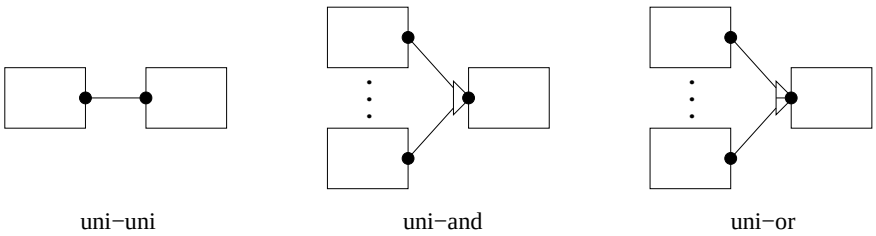

Fig. 1. Legal attachments

We now illustrate PADL by means of an example concerning a pipe-filter system. The system, which is depicted in Fig. 2 in accordance with the graphical notation, is composed of four identical filters and one pipe. Each filter acts as a service center of capacity two that is subject to failures and subsequent repairs. For each item processed by the upstream filter, the pipe forwards it to one of the three downstream filters according to the availability of free positions in their

buffers. If all the downstream filters have free positions, the choice is resolved nondeterministically.

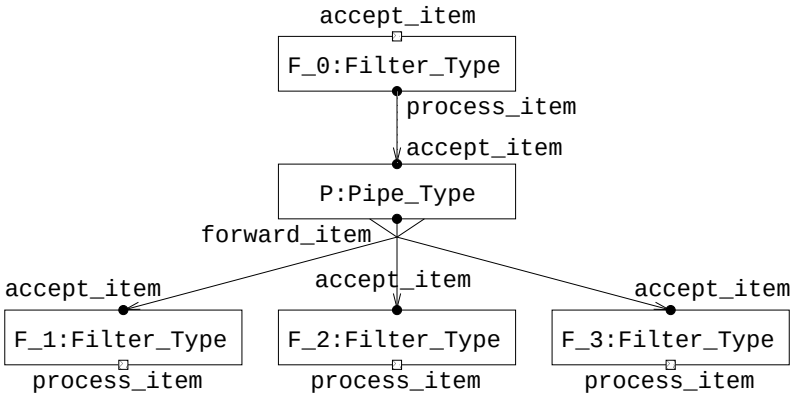

Fig. 2. Graphical description of `Pipe_Filter`

The pipe-filter system of Fig. 2 can be modeled with PADL as follows. First, we define the name of the architectural type:

```
ARCHI_TYPE Pipe_Filter
```

Second, we start the AET definition section of the PADL description by specifying the behavior and the interactions of the filter component type:

```
ARCHI_ELEM_TYPES

  ELEM_TYPE Filter_Type
    BEHAVIOR
      Filter_0 = accept_item . Filter_1 +
                 fail . repair . Filter_0
      Filter_1 = accept_item . Filter_2 +
                 process_item . Filter_0 +
                 fail . repair . Filter_1
      Filter_2 = process_item . Filter_1 +
                 fail . repair . Filter_2
    INPUT_INTERACTIONS
      UNI accept_item
    OUTPUT_INTERACTIONS
      UNI process_item
```

Initially (`Filter_0`), the filter waits for an item to arrive. When an item is already in the filter buffer (`Filter_1`), there are two possibilities: either another item arrives at the filter, or a previously arrived item finishes to be processed and is

sent out. Finally, when two items are already in the filter buffer (`Filter_2`), no more items can be accepted until one of the two previously arrived items finishes to be processed. In each of the three cases above, the filter can alternatively fail and be subsequently repaired. The action `accept_item` is declared to be an input uni-interaction, i.e. it can synchronize only with one output interaction of another AEI. The action `process_item`, instead, is declared to be an output uni-interaction, i.e. it can synchronize only with one input interaction of another AEI.

Third, we define the behavior and the interactions of the pipe component type:

```
ELEM_TYPE Pipe_Type
  BEHAVIOR
    Pipe = accept_item . forward_item . Pipe
  INPUT_INTERACTIONS
    UNI accept_item
  OUTPUT_INTERACTIONS
    OR forward_item
```

The pipe waits for an item, forwards it to one of several different destinations, then repeats this behavior. The fact that there may be several different destinations, and that the item is forwarded only to one of them, is witnessed by the declaration of `forward_item` as an or-interaction.

Fourth, we start the architectural topology section of the PADL description by declaring the instances of the previously defined AETs that compose the pipe-filter system of Fig. 2:

```
ARCHI_TOPOLOGY

  ARCHI_ELEM_INSTANCES
    F_0 : Filter_Type
    P   : Pipe_Type
    F_1 : Filter_Type
    F_2 : Filter_Type
    F_3 : Filter_Type
```

Fifth, we declare the architectural interactions, which can be used as global interfaces in the case in which the current architectural type is invoked in the definition of the behavior of a component type of a larger architectural type (hierarchical modeling):

```
ARCHI_INTERACTIONS
  F_0.accept_item
  F_1.process_item
  F_2.process_item
  F_3.process_item
```

Finally, we conclude the PADL description by specifying the attachments between the previously declared AEIs in order to reproduce the topology depicted in Fig. 2:

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

END
```

2.3 PADL Semantics

The semantics of a PADL specification is given by translation into PA. This translation is carried out in two steps. In the first step, the semantics of each AEI is defined to be the behavior of the corresponding AET projected onto its interactions. Such a projected behavior is obtained from the family of sequential PA terms representing the behavior of the AET by applying a hiding operator on all the actions that are not interactions. In this way, we abstract from all the internal details of the behavior of the AEI. In addition, the projected behavior must reflect the fact that an or-interaction can result in several distinct synchronizations. Therefore, every or-interaction is rewritten as a choice between as many indexed instances of uni-interactions as there are attachments involving the or-interaction.

Definition 1. *Let $\mathcal{A}$ be an architectural type and C be one of its AEIs with behavior E and interaction set $\mathcal{I}$. The semantics of C is defined by*

$$[\![C]\!] = \mathit{or\text{-}rewrite}(E)/(\mathit{Act} - \{\tau\} - \mathcal{I})$$

where $\mathit{or\text{-}rewrite}(E)$ is defined by structural induction as follows:

$$\begin{aligned}
\mathit{or\text{-}rewrite}(\underline{0}) &= \underline{0} \\
\mathit{or\text{-}rewrite}(a.G) &= \begin{cases} a.\mathit{or\text{-}rewrite}(G) & a\ \text{not an or–interaction} \\ \sum_{i=1}^{n} a_i.\mathit{or\text{-}rewrite}(G) & a\ \text{or–interaction with}\ n\ \text{attachs} \end{cases} \\
\mathit{or\text{-}rewrite}(G_1 + G_2) &= \mathit{or\text{-}rewrite}(G_1) + \mathit{or\text{-}rewrite}(G_2) \\
\mathit{or\text{-}rewrite}(A) &= A
\end{aligned}$$

■

For the pipe-filter system of Fig. 2 we have

$$\begin{aligned}
[\![\texttt{F_0}]\!] &= \texttt{Filter_0}/\{\texttt{fail}, \texttt{repair}\} \\
[\![\texttt{F_1}]\!] &= \texttt{Filter_0}/\{\texttt{fail}, \texttt{repair}\} \\
[\![\texttt{F_2}]\!] &= \texttt{Filter_0}/\{\texttt{fail}, \texttt{repair}\} \\
[\![\texttt{F_3}]\!] &= \texttt{Filter_0}/\{\texttt{fail}, \texttt{repair}\} \\
[\![\texttt{P}]\!] &= \mathit{or\text{-}rewrite}(\texttt{Pipe})
\end{aligned}$$

where $\mathit{or\text{-}rewrite}(\texttt{Pipe})$ is a constant `Pipe'` such that

```
Pipe' = accept_item . (forward_item_1 . Pipe' +
                       forward_item_2 . Pipe' +
                       forward_item_3 . Pipe')
```

It is worth observing that, in the semantics of the filters, the internal activities `fail` and `repair` have been abstracted away.

In the second step, the semantics of the architectural type is obtained by composing in parallel the semantics of its AEIs according to the specified attachments (the involved or-interactions need to be indexed). Recalled that the parallel composition operator is left associative, for the pipe-filter system of Fig. 2 we have

$$\begin{aligned}
[\![\texttt{Pipe_Filter}]\!] = {} & [\![\texttt{F_0}]\!][\texttt{process_item} \mapsto \texttt{a}] \parallel_{\{\texttt{a}\}} \\
& [\![\texttt{P}]\!][\texttt{accept_item} \mapsto \texttt{a}, \\
& \quad \texttt{forward_item_1} \mapsto \texttt{a_1}, \\
& \quad \texttt{forward_item_2} \mapsto \texttt{a_2}, \\
& \quad \texttt{forward_item_3} \mapsto \texttt{a_3}] \parallel_{\{\texttt{a_1}\}} \\
& \quad [\![\texttt{F_1}]\!][\texttt{accept_item} \mapsto \texttt{a_1}] \parallel_{\{\texttt{a_2}\}} \\
& \quad \quad [\![\texttt{F_2}]\!][\texttt{accept_item} \mapsto \texttt{a_2}] \parallel_{\{\texttt{a_3}\}} \\
& \quad \quad \quad [\![\texttt{F_3}]\!][\texttt{accept_item} \mapsto \texttt{a_3}]
\end{aligned}$$

The use of the relabeling operator is necessary to make the AEIs interact. As an example, `F_0` and `P` must interact via `process_item` and `accept_item`, which have different names. Since the parallel composition operator allows only equal actions to synchronize, in $[\![\texttt{Pipe\_Filter}]\!]$ each `process_item` action executed by $[\![\texttt{F\_0}]\!]$ and each `accept_item` action executed by $[\![\texttt{P}]\!]$ is relabeled to the same action `a`. In order to avoid interferences, it is important that `a` be a fresh action, i.e. an action occurring neither in $[\![\texttt{F\_0}]\!]$ nor in $[\![\texttt{P}]\!]$. Then a synchronization on `a` is forced between the relabeled versions of $[\![\texttt{F\_0}]\!]$ and $[\![\texttt{P}]\!]$ by means of operator $\parallel_{\{\texttt{a}\}}$.

In general, when accomplishing the second step, first of all we have to determine the number of fresh actions that we need in order to make the AEIs interact according to the attachments. To achieve that, we have to single out all the maximal sets of synchronizing 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. In the case of an or-interaction, we have as many maximal sets of synchronizing interactions as there are attachments involving the or-interaction; each of such sets comprises the uni-interaction involved in the attachment and the uni-interaction obtained by indexing the or-interaction. 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.

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 interactions, we construct a set $\mathcal{S}(C_1, \ldots, C_n)$ composed of as many fresh actions as there are maximal sets of synchronizing 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 action 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. Based on these relabeling functions that prepare the AEIs to interact, we now define two semantics for C_i restricted to its local interactions attached to local interactions of $C_1, \ldots, C_n$. The closed semantics will be used for deadlock freedom verification purposes. It abstracts from the architectural interactions of C_i as these must not come into play when checking for deadlock freedom. Since the open semantics will be used instead in the definition of the semantics of an architectural type, it does not abstract from the architectural interactions of C_i as these must be observable. If C_i has no architectural interactions, then the two semantics coincide.

Definition 2. *The closed and the open interacting semantics of C_i restricted to $C_1, \ldots, C_n$ are defined by*

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

Finally, we define the closed and the open interacting semantics of $C_1, \ldots, C_n$ by putting in parallel the closed and the open interacting semantics of each of the considered AEIs, respectively. To do that, we need to define the synchronization sets. Let us preliminarily 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:

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

Recalled that the parallel composition operator is left associative, the synchronization set between the interacting semantics of C_1 and C_2 is given by $\mathcal{S}(C_1, C_2; C_1, \ldots, C_n)$, the synchronization set between the interacting semantics of C_2 and C_3 is given by $\mathcal{S}(C_1, C_3; C_1, \ldots, C_n) \cup \mathcal{S}(C_2, C_3; C_1, \ldots, C_n)$, and so on.

Definition 3. *The closed and the open interacting semantics of $C_1, \ldots, C_n$ are defined by*

$$\begin{array}{l} [\![C_1, \ldots, C_n]\!]^{\mathrm{c}} = [\![C_1]\!]^{\mathrm{c}}_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C_1,C_2;C_1,\ldots,C_n)} \\ \qquad [\![C_2]\!]^{\mathrm{c}}_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C_1,C_3;C_1,\ldots,C_n) \cup \mathcal{S}(C_2,C_3;C_1,\ldots,C_n)} \cdots \\ \qquad\quad \cdots \parallel_{\cup_{i=1}^{n-1} \mathcal{S}(C_i,C_n;C_1,\ldots,C_n)} [\![C_n]\!]^{\mathrm{c}}_{C_1,\ldots,C_n} \\ [\![C_1, \ldots, C_n]\!]^{\mathrm{o}} = [\![C_1]\!]^{\mathrm{o}}_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C_1,C_2;C_1,\ldots,C_n)} \\ \qquad [\![C_2]\!]^{\mathrm{o}}_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C_1,C_3;C_1,\ldots,C_n) \cup \mathcal{S}(C_2,C_3;C_1,\ldots,C_n)} \cdots \\ \qquad\quad \cdots \parallel_{\cup_{i=1}^{n-1} \mathcal{S}(C_i,C_n;C_1,\ldots,C_n)} [\![C_n]\!]^{\mathrm{o}}_{C_1,\ldots,C_n} \end{array}$$ ■

Definition 4. *The semantics of an architectural type $\mathcal{A}$ whose AEIs are $C_1, \ldots, C_n$ is defined by $[\![\mathcal{A}]\!] = [\![C_1, \ldots, C_n]\!]^{\mathrm{o}}$.* ■

3 Deadlock Freedom Verification

The use of PADL for modeling large software systems represents a step towards bridging the gap between the rigorous view of difficult-to-use formal methods and the practical view of the software architect. However, if we want such an approach to be perceived as sufficiently appealing and profitable in practice,

it must be accompanied by scalable and simple-to-use techniques both for the automatic detection of architectural mismatches and for the identification of their origins.

Among the several different architectural mismatches that can be encountered in the design process, in this paper we concentrate on deadlock. As mentioned in Sect. 1, two different architectural checks, called compatibility check and interoperability check, have been developed in [3] that deal with deadlock related architectural mismatches for two different topologies: acyclic architectural types and ring architectural types.

In this section, we present a general architectural check, which can be applied to any architectural type independently of its topology and provides a sufficient condition for deadlock freedom. To this purpose, we preliminarily recall from [3] the notion of reduced flow graph as well as the notion of compatibility check and we introduce a slight variant of the interoperability check. Based on these definitions, we then propose a novel technique for verifying deadlock freedom at the architectural level of design for systems with an arbitrary topology.

3.1 Reduced Flow Graph

When applying the deadlock related architectural checks to PADL descriptions of architectural types, as seen in [3] we can safely abstract from the direction of the information flow and from the multiplicity of the attachments between pairs of AEIs. As a consequence, an architectural type is classified as having an acyclic topology or a cyclic topology based on a modification of its graphical representation. The result of such a modification, called reduced flow graph, collapses all the directed edges between two boxes into a single, indirect edge.

3.2 Compatibility Check for Acyclic Topologies

The main principle underlying the compatibility check of [3] is based on the observation that 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 and denoted by $\mathcal{B}_K$. The absence of cycles guarantees that $C_1, \ldots, C_n$ cannot directly communicate with each other. Therefore, the absence of deadlock can be investigated by analyzing the interactions between the center K of the star topology and the AEIs constituting the border of the star topology. The important result that can be derived is that verifying deadlock freedom for the whole architectural type reduces to checking the local interactions within each of the constituent star topologies.

The architectural compatibility check for a star topology with center AEI K attached to AEIs $C_1, \ldots, C_n$ works as follows. The intuition is that K is compatible with C_i if the potential interactions of K with the star topology components are not altered when attaching C_i to K. Formally, we verify whether the closed interacting semantics of K with respect to the star topology, namely $[\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}$, is weakly bisimulation equivalent to the parallel composition of the

closed interacting semantics of K and C_i. If this holds for any C_i of the star topology, then the interactions of K cannot be limited by the behavior of its neighbors.

Definition 5. *Given an architectural type $\mathcal{A}$, let $C_1, \ldots, C_n$ be the AEIs attached to an AEI K in $\mathcal{A}$. C_i is said to be compatible with K iff*

$$\llbracket K \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{\mathcal{S}(K;K,\mathcal{B}_K)} \llbracket C_i \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K} \approx_{\mathrm{B}} \llbracket K \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K}$$

■

In a star topology, the compatibility between the center K and each C_i attached to K provides a sufficient condition for deadlock freedom in case K is deadlock free. Therefore, the deadlock freedom result for the whole star topology is obtained by simply applying peer-to-peer checks between its constituents. The main result saying that the absence of deadlock scales to the whole acyclic architectural type in case all the star topologies are deadlock free, is summarized by the following theorem [3].

Theorem 1. (Compatibility) *Let $\mathcal{A}$ be an acyclic architectural type. If the semantics of each AEI of $\mathcal{A}$ – with the architectural interactions being hidden – is deadlock free and every AEI of $\mathcal{A}$ is compatible with each AEI attached to it, then $\llbracket \mathcal{A} \rrbracket$ is deadlock free.* ■

3.3 Interoperability Check for Ring Topologies

Ensuring deadlock freedom for cyclic architectural types cannot be achieved by employing the peer-to-peer compatibility check described above, as there may be further causes of architectural mismatches due to the cyclic nature of the topology. To this aim, the interoperability condition presented in [3] is used to verify deadlock freedom in the presence of cycles. The intuition behind the interoperability check is almost the same as that of the compatibility check. Informally, given a cycle formed by the AEIs $C_1, \ldots, C_n$, if the potential local interactions of a given C_i are not altered when attaching C_i to the cycle, then the behavior of the cycle is the same as that expected by C_i and we say that C_i interoperates with the cycle. If there exists such a C_i within the cycle and C_i is deadlock free, then the cycle is deadlock free. Hence, with respect to the compatibility notion, here the minimal group of AEIs to be included in each check is given by all the AEIs $C_1, \ldots, C_n$ forming the cycle. This is because any AEI within the cycle could be responsible for limiting the local interactions of C_i with its neighbors.

In the following, given an architectural type $\mathcal{A}$ whose AEIs are $K_1, \ldots, K_m$, by abuse of notation we will use the abbreviation $\mathcal{A}$ to stand for $K_1, \ldots, K_m$. For instance, $\llbracket K \rrbracket^{\mathrm{c}}_{\mathcal{A}}$ stands for $\llbracket K \rrbracket^{\mathrm{c}}_{K_1,\ldots,K_m}$ and $\mathcal{S}(K;\mathcal{A})$ stands for $\mathcal{S}(K;K_1,\ldots,K_m)$.

Definition 6. *Let $\mathcal{A}$ be an architectural type and $C_1, \ldots, C_n$ be some of its AEIs. The closed interacting semantics of $C_1, \ldots, C_n$ with respect to $\mathcal{A}$ is defined by*

$$\begin{aligned}\llbracket C_1, \ldots, C_n \rrbracket^{\mathrm{c}}_{\mathcal{A}} = \llbracket C_1 \rrbracket^{\mathrm{c}}_{\mathcal{A}} \parallel_{\mathcal{S}(C_1,C_2;\mathcal{A})} & \\ \llbracket C_2 \rrbracket^{\mathrm{c}}_{\mathcal{A}} \parallel_{\mathcal{S}(C_1,C_3;\mathcal{A}) \cup \mathcal{S}(C_2,C_3;\mathcal{A})} \cdots & \\ \cdots \parallel_{\cup_{i=1}^{n-1} \mathcal{S}(C_i,C_n;\mathcal{A})} \llbracket C_n \rrbracket^{\mathrm{c}}_{\mathcal{A}} &\end{aligned}$$

■

The following definition formalizes the notion of interoperability as described above. Note that the behavior of a single AEI in the cycle is compared with the behavior of the whole cycle projected on the local interactions of that specific AEI.

Definition 7. *Given an architectural type $\mathcal{A}$, let $C_1, \ldots, C_n$ be AEIs forming a cycle in the reduced flow graph of $\mathcal{A}$. C_i is said to interoperate with $C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n$ iff*

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

■

We point out that the interoperability notion of [3] is slightly different from that of Def. 7. The former compares the parallel composition of the closed interacting semantics of $C_1, \ldots, C_n$ projected on the interactions with C_i only and the closed interacting semantics of C_i projected on the interactions with $C_1, \ldots, C_n$. Instead, in Def. 7 all the local interactions of C_i are left visible. As we shall see in Sect. 3.4, this is needed if we want the results of the interoperability check to scale in the case of cyclic architectural types that are not rings. Obviously, the two notions of interoperability coincide in case the architectural type is a ring.

Before introducing the interoperability theorem, with respect to [3] we add the notion of frontier, which is useful to define a ring topology and also to prove the main result of this paper in Sect. 3.4.

Definition 8. *Given an architectural type $\mathcal{A}$, let $C_1, \ldots, C_n$ be some of its AEIs. The frontier of $C_1, \ldots, C_n$ is the unique subset $\mathcal{F}_{C_1,\ldots,C_n}$ of $\{C_1, \ldots, C_n\}$ such that $C_i \in \mathcal{F}_{C_1,\ldots,C_n}$ iff C_i is attached to AEI $K \notin \{C_1, \ldots, C_n\}$.* ■

Definition 9. *Let $\mathcal{A}$ be an architectural type. $\mathcal{A}$ is said to be a ring formed by the AEIs $C_1, \ldots, C_n$ iff $\mathcal{F}_{C_1,\ldots,C_n} = \emptyset$ and for each proper subset $\{C'_1, \ldots, C'_{n'}\}$ of $\{C_1, \ldots, C_n\}$, $C'_1, \ldots, C'_{n'}$ do not form a cycle in the reduced flow graph of $\mathcal{A}$.* ■

Theorem 2. (Interoperability) *Let the architectural type $\mathcal{A}$ be a ring formed by the AEIs $C_1, \ldots, C_n$. If there exists C_i such that $[\![C_i]\!]^{\mathrm{c}}_{\mathcal{A}}$ is deadlock free and C_i interoperates with $C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n$, then $[\![\mathcal{A}]\!]$ is deadlock free.* ■

3.4 General Check for Arbitrary Topologies

While the architectural compatibility check scales from star topologies to arbitrary acyclic topologies, the architectural interoperability check does not scale from ring topologies to arbitrary cyclic topologies. This is because of 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 architectural interoperability check applied to a cycle of AEIs $C_1, \ldots, C_n$ does not provide a sufficient condition for deadlock freedom if the cycle is such that some C_i interacts with some AEI K that is not in $C_1, \ldots, C_n$. In other words, if the frontier of the cycle is not empty, then the

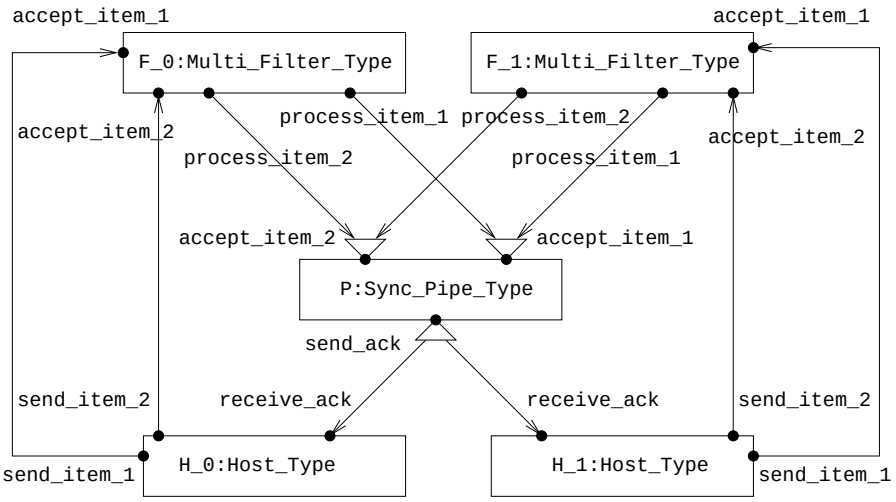

Fig. 3. Graphical description of `Feedback_PF`

interoperability condition is not enough to decide the deadlock freedom. Assume, e.g., that it is possible to find a C_i in the cycle such that its interactions are not affected by the behavior of the other AEIs of the cycle. Even if C_i interoperates with the cycle, nothing can be deduced about the influence of other components of the architectural topology upon the cycle in case some AEIs of the cycle interact with some AEIs outside the cycle. This is because, when checking the interoperability condition for C_i, we abstract away from the interactions that attach the other components of the cycle to AEIs external to the cycle.

Let us consider, e.g., the system depicted in Fig. 3 and specified with PADL in Table 2. The system, called `Feedback_PF`, is composed of two hosts, two filters of capacity one, and a pipe with feedback. Each host forms a cycle with its dedicated filter and the pipe. In particular, host $\mathtt{H}_0$ generates two types of items (called type 1 and type 2), which are sent to filter $\mathtt{F}_0$, which in turn processes and passes the received items to the pipe (similarly for host $\mathtt{H}_1$). Pipe `P` is willing to receive an item of type 1 from filter $\mathtt{F}_0$ if and only if filter $\mathtt{F}_1$ is ready to send an item of the same type too. The reception of these two items from both filters is synchronized (see the uni-and attachments between the two filters and the pipe in Fig. 3). Upon the synchronized reception of the two items of type 1, pipe `P` sends an acknowledgement to each host (see the uni-and attachments between the pipe and the two hosts in Fig. 3). We can argue similarly for items of type 2. Hence, pipe `P` can receive items if both filters (*i*) are ready to send an item and (*ii*) agree on the type of item to be processed. As we shall see, such a behavior of the pipe potentially causes a deadlock that cannot be detected through the interoperability check. Consider, e.g., the scenario where filter $\mathtt{F}_0$ processes an item of type 1, while filter $\mathtt{F}_1$ processes an item of type 2. The cycle composed of

Table 2. PADL description of `Feedback_PF`

```
ARCHI_TYPE Feedback_PF

ARCHI_ELEM_TYPES
   ELEM_TYPE Host_Type
      BEHAVIOR Host = send_item_1 . receive_ack . Host +
                      send_item_2 . receive_ack . Host
      INPUT_INTERACTIONS  UNI receive_ack
      OUTPUT_INTERACTIONS UNI send_item_1, send_item_2
   ELEM_TYPE Multi_Filter_Type
      BEHAVIOR Filter   = accept_item_1 . Filter' +
                          accept_item_2 . Filter'' +
                          fail . repair . Filter
               Filter'  = process_item_1 . Filter +
                          fail . repair . Filter'
               Filter'' = process_item_2 . Filter +
                          fail . repair . Filter''
      INPUT_INTERACTIONS  UNI accept_item_1, accept_item_2
      OUTPUT_INTERACTIONS UNI process_item_1, process_item_2
   ELEM_TYPE Sync_Pipe_Type
      BEHAVIOR Pipe  = accept_item_1 . send_ack . Pipe +
                       accept_item_2 . send_ack . Pipe
      INPUT_INTERACTIONS  AND accept_item_1, accept_item_2
      OUTPUT_INTERACTIONS AND send_ack

ARCHI_TOPOLOGY
   ARCHI_ELEM_INSTANCES H0, H1 : Host_Type
                        F0, F1 : Multi_Filter_Type
                        P      : Sync_Pipe_Type
   ARCHI_INTERACTIONS
   ARCHI_ATTACHMENTS FROM H0.send_item_1    TO F0.accept_item_1
                     FROM H0.send_item_2    TO F0.accept_item_2
                     FROM H1.send_item_1    TO F1.accept_item_1
                     FROM H1.send_item_2    TO F1.accept_item_2
                     FROM F0.process_item_1 TO P.accept_item_1
                     FROM F0.process_item_2 TO P.accept_item_2
                     FROM F1.process_item_1 TO P.accept_item_1
                     FROM F1.process_item_2 TO P.accept_item_2
                     FROM P.send_ack        TO H0.receive_ack
                     FROM P.send_ack        TO H1.receive_ack

END
```

$\mathtt{H}_0$, $\mathtt{F}_0$, and $\mathtt{P}$ deadlocks since $\mathtt{H}_0$ waits for an acknowledgement from $\mathtt{P}$, $\mathtt{F}_0$ waits for delivering the item of type 1 to the pipe, which instead waits for an item of the same type from $\mathtt{F}_1$. On the other hand, filter $\mathtt{F}_1$ is blocked since it is trying to send an item of type 2 to $\mathtt{P}$ and, as a consequence, host $\mathtt{H}_1$ is blocked until the reception of an acknowledgement that pipe $\mathtt{P}$ cannot send. However, it can be verified that $\mathtt{H}_0$ interoperates with $\mathtt{F}_0$ and $\mathtt{P}$, and $\mathtt{H}_1$ interoperates with $\mathtt{F}_1$ and $\mathtt{P}$. More formally, we have that, e.g., $[\![\mathtt{H}_0]\!]^{\mathtt{c}}_{\mathtt{Feedback\_PF}}$ is weakly bisimulation equivalent to the closed interacting semantics of AEIs $\mathtt{H}_0$, $\mathtt{F}_0$, and $\mathtt{P}$ with respect to $\mathtt{Feedback\_PF}$. As can easily be seen, in both cases we abstract away from the local interactions of $\mathtt{P}$ with $\mathtt{F}_1$, which is not in the cycle. Therefore, we cannot verify the influence of the cycle behavior upon the interaction between $\mathtt{P}$ and $\mathtt{F}_1$ and, as a consequence, we cannot reveal the mismatch. Key to a successful detection of the deadlock is an interoperability check applied to $\mathtt{P}$, whose interactions with both cycles cause the troublesome behavior described above.

We now show that, even in an arbitrary architectural topology like the one in Fig. 3, it is possible to verify the absence of deadlock by analyzing some specific local interactions of its AEIs. In the following theorem, deadlock freedom is guaranteed for an arbitrary architectural type under three assumptions. First, every AEI must be deadlock free. Second, every AEI must be compatible with each AEI attached to it. This ensures deadlock freedom for acyclic topologies. Third, if the architectural type has a cyclic topology, then there exists a cycle covering strategy (Def. 11) such that two constraints are satisfied, which are concerned with a set of intersecting cycles called a cyclic border (Def. 10). The first constraint requires that, if the architectural type is formed by a single cyclic border with empty frontier, then it must contain an AEI that interoperates with the other AEIs in the cyclic border (in analogy with the interoperability check for ring topologies). The second constraint requires that every AEI K in the frontier of any cyclic border must interoperate with all the other AEIs belonging to the cyclic border. This ensures a deadlock free combination of cyclic borders and acyclic portions of the topology.

Definition 10. *Given an architectural type $\mathcal{A}$, let K be one of its AEIs such that K is in (at least) one cycle in the reduced flow graph of $\mathcal{A}$. The set of all the AEIs involved with K in (at least) one cycle of the reduced flow graph of $\mathcal{A}$, called the cyclic border of K, is defined by $\mathcal{CB}^{\mathcal{A}}_K = \{K\} \cup \{H \mid \exists C_1, \ldots, C_n : K, H, C_1, \ldots, C_n$ form a cycle in the reduced flow graph of $\mathcal{A}\}$.* ■

Definition 11. *Given a cyclic architectural type $\mathcal{A}$, a cycle covering strategy is defined by the following algorithm:*

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

The application of a cycle covering strategy to a cyclic architectural type $\mathcal{A}$ generates a set involving all the AEIs in the cycles of the reduced flow graph of $\mathcal{A}$, which contains the cyclic borders considered by the algorithm.

Lemma 1. *Given a cyclic architectural type $\mathcal{A}$ and a cycle covering strategy that originates the set $\{\mathcal{CB}^{\mathcal{A}}_{K_1}, \ldots, \mathcal{CB}^{\mathcal{A}}_{K_n}\}$, then the two following conditions hold:*

1. *For any pair of different cyclic borders $\mathcal{CB}^{\mathcal{A}}_{K_i}, \mathcal{CB}^{\mathcal{A}}_{K_j} \in \{\mathcal{CB}^{\mathcal{A}}_{K_1}, \ldots, \mathcal{CB}^{\mathcal{A}}_{K_n}\}$, $\mathcal{CB}^{\mathcal{A}}_{K_i}$ can be directly attached to $\mathcal{CB}^{\mathcal{A}}_{K_j}$ in two different ways only:*
   - *I. They interact through a single, shared AEI K.*
   - *II. They do not share any AEI, but they interact through attachments between a single AEI H of $\mathcal{CB}^{\mathcal{A}}_{K_i}$ and a single AEI H' of $\mathcal{CB}^{\mathcal{A}}_{K_j}$.*
2. *If we replace each $\mathcal{CB}^{\mathcal{A}}_{K_i} = \{H_1, \ldots, H_l\}$ with an AEI that is isomorphic to*
$$[\![H_1, \ldots, H_l]\!]^{\mathrm{c}}_{\mathcal{A}} / (Act - \{\tau\} - \bigcup_{H_j \in \mathcal{F}_{H_1,\ldots,H_l}} \mathcal{S}(H_j; \mathcal{A}))$$
then the obtained architectural topology is acyclic.

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

As far as condition 1.II is concerned, assume that there exists another attachment between an AEI H'' of $\mathcal{CB}^{\mathcal{A}}_{K_i}$ and an AEI H''' of $\mathcal{CB}^{\mathcal{A}}_{K_j}$. Then the reduced flow graph of $\mathcal{A}$ would contain a cycle including $K_i, H, H', K_j, H''', H''$, thus contradicting the hypothesis that $\mathcal{CB}^{\mathcal{A}}_{K_i}$ is the cyclic border of K_i. On the other hand, if there exists another attachment between an AEI H'' of $\mathcal{CB}^{\mathcal{A}}_{K_i}$ and H', then the reduced flow graph of $\mathcal{A}$ would contain a cycle including K_i, H, H', H'', thus contradicting the hypothesis that $\mathcal{CB}^{\mathcal{A}}_{K_i}$ and $\mathcal{CB}^{\mathcal{A}}_{K_j}$ do not share any AEI. We can argue similarly in case of an attachment between an AEI H'' of $\mathcal{CB}^{\mathcal{A}}_{K_j}$ and H.

As far as condition 2 is concerned, the proof is a straightforward consequence of condition 1 and of the maximality of each cyclic border. ■

Theorem 3. *Let $\mathcal{A}$ be an architectural type with an arbitrary topology. Suppose that the following conditions hold:*

1. *For every AEI K in $\mathcal{A}$, $[\![K]\!]^{\mathrm{c}}_{\mathcal{A}}$ is deadlock free.*
2. *Every AEI of $\mathcal{A}$ is compatible with each AEI attached to it.*
3. *If $\mathcal{A}$ is cyclic, then there exists a set of cyclic borders generated by a cycle covering strategy such that:*
   - *I. If the set has a single cyclic border $\{C_1, \ldots, C_n\}$ such that $\mathcal{F}_{C_1,\ldots,C_n} = \emptyset$, then there exists C_i that interoperates with $C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n$.*
   - *II. Otherwise, for every cyclic border $\{C_1, \ldots, C_n\}$ in the set, we have that for each $C_i \in \mathcal{F}_{C_1,\ldots,C_n}$, C_i interoperates with $C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n$.*

Then $[\![\mathcal{A}]\!]$ is deadlock free.

Proof. *We proceed by induction on the number m of cycles in the reduced flow graph of $\mathcal{A}$. As far as the induction base is concerned, if $m = 0$, then $\mathcal{A}$ is acyclic and the proof, by 1 and 2, is a straightforward consequence of the compatibility theorem.*

Let the result hold for a certain $m \geq 0$ and consider an architectural type $\mathcal{A}$ satisfying 1, 2, and 3, whose reduced flow graph has $m+1$ cycles. Let $\{\mathcal{CB}^{\mathcal{A}}_K, \mathcal{CB}^{\mathcal{A}}_{K_1}, \ldots, \mathcal{CB}^{\mathcal{A}}_{K_n}\}$ be the set of cyclic borders originated by the cycle covering strategy of 3 and, by virtue of condition 2 of Lemma 1, let $\mathcal{CB}^{\mathcal{A}}_K = \{C_1, \ldots, C_n\}$ be a cyclic border that directly interacts with at most one cyclic border in the set. Now we replace the AEIs $C_1, \ldots, C_n$ with a new AEI C such that its behavior is isomorphic to $[\![C_1, \ldots, C_n]\!]^{\mathrm{c}}_{\mathcal{A}}/(Act - \{\tau\} - \bigcup_{C_j \in \mathcal{F}_{C_1,\ldots,C_n}} \mathcal{S}(C_j;\mathcal{A}))$, thus obtaining an architectural type $\mathcal{A}'$ such that:

- *C preserves 1. In fact, by 3, there exists C_i such that*
$$[\![C_1, \ldots, C_n]\!]^{\mathrm{c}}_{\mathcal{A}}/(Act - \{\tau\} - \mathcal{S}(C_i;\mathcal{A})) \approx_{\mathrm{B}} [\![C_i]\!]^{\mathrm{c}}_{\mathcal{A}}$$
from which we derive that $[\![C_1, \ldots, C_n]\!]^{\mathrm{c}}_{\mathcal{A}}/(Act - \{\tau\} - \mathcal{S}(C_i;\mathcal{A}))$ is deadlock free because so is $[\![C_i]\!]^{\mathrm{c}}_{\mathcal{A}}$ due to 1. Therefore, we also have that $[\![C_1, \ldots, C_n]\!]^{\mathrm{c}}_{\mathcal{A}}/(Act - \{\tau\} - \bigcup_{C_j \in \mathcal{F}_{C_1,\ldots,C_n}} \mathcal{S}(C_j;\mathcal{A}))$ is deadlock free.
- *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*
$$[\![C_i]\!]^{\mathrm{c}}_{C_i,\mathcal{B}_{C_i}} \|_{\mathcal{S}(C_i;C_i,\mathcal{B}_{C_i})} [\![H]\!]^{\mathrm{c}}_{C_i,\mathcal{B}_{C_i}} \approx_{\mathrm{B}} [\![C_i]\!]^{\mathrm{c}}_{C_i,\mathcal{B}_{C_i}}$$
from which it follows that
$$[\![C_i]\!]^{\mathrm{c}}_{\mathcal{A}} \|_{\mathcal{S}(C_i;\mathcal{A})} [\![H]\!]^{\mathrm{c}}_{C_i,\mathcal{B}_{C_i}} \approx_{\mathrm{B}} [\![C_i]\!]^{\mathrm{c}}_{\mathcal{A}}$$
Since $\approx_{\mathrm{B}}$ is a congruence with respect to the parallel composition operator,
$$[\![C]\!]^{\mathrm{c}}_{\mathcal{A}'} \|_{\mathcal{S}(C;\mathcal{A}')} [\![H]\!]^{\mathrm{c}}_{C,\mathcal{B}_C} \approx_{\mathrm{B}} [\![C]\!]^{\mathrm{c}}_{\mathcal{A}'}$$
because we hide interactions that are not attached to H (only C_i can be attached to H otherwise $\mathcal{CB}^{\mathcal{A}}_K$ would not be a cyclic border), from which it follows that
$$[\![C]\!]^{\mathrm{c}}_{C,\mathcal{B}_C} \|_{\mathcal{S}(C;C,\mathcal{B}_C)} [\![H]\!]^{\mathrm{c}}_{C,\mathcal{B}_C} \approx_{\mathrm{B}} [\![C]\!]^{\mathrm{c}}_{C,\mathcal{B}_C}$$
Similarly, it can be shown that
$$[\![H]\!]^{\mathrm{c}}_{H,\mathcal{B}_H} \|_{\mathcal{S}(H;H,\mathcal{B}_H)} [\![C]\!]^{\mathrm{c}}_{H,\mathcal{B}_H} \approx_{\mathrm{B}} [\![H]\!]^{\mathrm{c}}_{H,\mathcal{B}_H}$$

- *If $\mathcal{A}'$ is cyclic, then 3 is preserved. In fact, let $\{\overline{\mathcal{CB}}^{\mathcal{A}'}_{K_1}, \ldots, \overline{\mathcal{CB}}^{\mathcal{A}'}_{K_n}\}$ be a new set of cyclic borders for $\mathcal{A}'$ obtained from the cyclic borders $\mathcal{CB}^{\mathcal{A}}_{K_1}, \ldots, \mathcal{CB}^{\mathcal{A}}_{K_n}$ of the old set for $\mathcal{A}$ by replacing every occurrence of $C_1, \ldots, C_n$ with C. Every cyclic border in the new set that does not include C has a corresponding isomorphic cyclic border in the old set. On the other hand, if we take in the new set a cyclic border formed by the AEIs $H_1, \ldots, H_l, C$, then the old set 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 condition 1 of Lemma 1. By virtue of 3.II,*
$$[\![C_i]\!]^{\mathrm{c}}_{\mathcal{A}} \approx_{\mathrm{B}} [\![H_1, \ldots, H_l, C_i]\!]^{\mathrm{c}}_{\mathcal{A}}/(Act - \{\tau\} - \mathcal{S}(C_i;\mathcal{A}))$$
Since $\approx_{\mathrm{B}}$ is a congruence with respect to the parallel composition operator,

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

because we hide interactions that do not occur in C. Thus, 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

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

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

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

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

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

because we hide interactions that do not occur in H_j.

– *The reduced flow graph of $\mathcal{A}'$ has at most m cycles.*

Then, by the induction hypothesis it follows that $[\![\mathcal{A}']\!]$ is deadlock free, from which we derive that $[\![\mathcal{A}]\!]$ is deadlock free because

$$[\![\mathcal{A}']\!] = [\![\mathcal{A}]\!] / (\bigcup_{C_i \notin \mathcal{F}_{C_1,\ldots,C_n}} \mathcal{S}(C_i; \mathcal{A}) - \bigcup_{C_i \in \mathcal{F}_{C_1,\ldots,C_n}} \mathcal{S}(C_i; \mathcal{A}))$$

■

We point out that a violation of one of the conditions of Thm. 3 does not imply that the architectural type can deadlock, but reveals the presence of some kind of mismatch in a specific portion of the topology. Diagnostic information can be inferred as explained in [3].

As far as the example of Table 2 is concerned, let us consider the set of cyclic borders $\{\mathcal{CB}^{\texttt{Feedback\_PF}}_{\texttt{H}_0}, \mathcal{CB}^{\texttt{Feedback\_PF}}_{\texttt{H}_1}\}$, obtained by applying a cycle covering strategy that does not pick up `P`. It can be verified that `P`, which represents the frontier for both cyclic borders $\{\texttt{H}_0, \texttt{F}_0, \texttt{P}\}$ and $\{\texttt{H}_1, \texttt{F}_1, \texttt{P}\}$, interoperates with neither $\texttt{H}_0$ and $\texttt{F}_0$, nor $\texttt{H}_1$ and $\texttt{F}_1$. For instance, the closed interacting semantics of $\texttt{H}_1, \texttt{F}_1, \texttt{P}$, computed with respect to `Feedback_PF` and projected on the local interactions of `P`, expresses the fact that the type of the item that the pipe can accept depends on the type chosen by $\texttt{F}_1$. Instead, the closed interacting semantics of `P` expresses the fact that the pipe is always ready to accept items of both types. Therefore, the two semantics cannot be weakly bisimulation equivalent and, as a consequence, the system has a potential mismatch that, as we have seen, in practice causes a deadlock.

4 Conclusion

In this paper we have presented a novel technique for deadlock freedom verification at the architectural level of design, which is independent of the architectural topology, thus overcoming the limitations of the techniques previously appeared in the literature. Applying such a technique is more convenient – for efficiency reasons and diagnostic purposes – than checking the whole system for deadlock freedom. On the efficiency side, the software architect is saved from generating

the state space associated with the whole system, which could be composed of millions of states for large software architectures. Instead, two checks are applied. The former check is a compatibility check, which reduces to compare the semantics of any AEI C with the semantics of the parallel composition of C and any K attached to C. The latter check is a variant of the interoperability check applied to each AEI K belonging to the frontier of a specific cyclic border. Such a check reduces to compare the semantics of K with the semantics of the cyclic border that includes K. It is worth noting that, for each check, the projection on the local interactions of a single AEI, which are the only observable interactions, offers the possibility of a compositional construction of the considered state spaces in a minimized form with respect to $\approx_{\mathrm{B}}$. This ensures a good degree of scalability in the average case. Concerning future research, we would like to investigate whether it is possible to further enhance the generality of the developed technique, passing from a specific property – deadlock freedom – to arbitrary properties expressed in some logic.

References

1. G.D. Abowd, R. Allen, and D. Garlan, *"Formalizing Style to Understand Descriptions of Software Architecture"*, in ACM Trans. on Software Engineering and Methodology 4:319-364, 1995.
2. R. Allen and D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM Trans. on Software Engineering and Methodology 6:213-249, 1997.
3. 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.
4. 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.
5. 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.
6. P. Inverardi and A.L. Wolf, *"Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model"*, in IEEE Trans. on Software Engineering 21:373-386, 1995.
7. 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.
8. 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.
9. R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989.
10. D.E. Perry and A.L. Wolf, *"Foundations for the Study of Software Architecture"*, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992.
11. M. Shaw and D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline"*, Prentice Hall, 1996.