Architecting Families of Software Systems with Process Algebras

MARCO BERNARDO
University of Urbino - Italy
and
PAOLO CIANCARINI and LORENZO DONATIELLO
University of Bologna - Italy

Software components can give rise to several kinds of architectural mismatches when assembled together in order to form a software system. A formal description of the architecture of the resulting component based software system may help to detect such architectural mismatches and to single out the components that cause the mismatches. In this paper we concentrate on deadlock related architectural mismatches arising from three different causes that we identify: incompatibility between two components due to a single interaction, incompatibility between two components due to the combination of several interactions, and lack of interoperability among a set of components forming a cyclic topology. We develop a process algebra based architectural description language called PADL, which deals with all the three causes through an architectural compatibility check and an architectural interoperability check relying on standard observational equivalences. The adequacy of the architectural compatibility check is assessed on a compressing proxy system, while the adequacy of the architectural interoperability check is assessed on a cruise control system. We then address the issue of scaling the architectural compatibility and interoperability checks to architectural styles through an extension of PADL. The formalization of an architectural style is complicated by the presence of two degrees of freedom within the set of instances of the style: variability of the internal behavior of the components and variability of the topology formed by the components. As a first step towards the solution of the problem, we propose an intermediate abstraction called architectural type, whose instances differ only for the internal behavior of their components. We define an efficient architectural conformity check based on a standard observational equivalence to verify whether an architecture is an instance of an architectural type. We show that all the architectures conforming to the same architectural type possess the same compatibility and interoperability properties.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; D.2.11 [**Software Engineering**]: Software Architectures—*languages*; *patterns*

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Architectural mismatch detection, architectural styles, process algebras, software architectures

Authors address: M. Bernardo, Univ. di Urbino, Ist. di Scienze e Tecnologie dell'Informazione, Piazza della Repubblica 13, 61029 Urbino, Italy; P. Ciancarini and L. Donatiello, Univ. di Bologna, Dip. di Scienze dell'Informazione, Mura A. Zamboni 7, 40127 Bologna, Italy.

This work has been supported by Progetto MURST Cofinanziato "Saladin".

1. INTRODUCTION

As recognized a long time ago, structuring a large collection of software components to form a software system is an essentially distinct and different intellectual activity from that of constructing the individual software components [DeRemer and Kron 1976]. The software architecture level of design enables us to cope with the increasing size and complexity of the component based software systems during the early stage of their development [Perry and Wolf 1992; Shaw and Garlan 1996]. To achieve this, the design focus is turned from algorithmic and data structure related issues 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 [Allen and Garlan 1997; Magee et al. 1995], Z [Abowd et al. 1995], and the CHAM [Inverardi and Wolf 1995].

The formal description of the software architecture of a complex system serves two purposes. First and foremost is making available a precise document describing the structure of the system to all the people involved in the design, implementation, and maintainance of the system itself. The second one is concerned with the possibility of analyzing the properties of the system at the architectural level. This allows for the early detection of errors, thus resulting in time and money saving. This paper is about the latter purpose.

Nowadays software systems are typically made out of numerous components whose behavior is individually well known. Typical examples are embedded systems and web centered applications. Thus, 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 example is deadlock: starting from deadlock free components, the designer constructs a system that can deadlock. This example is just an instance of a more general architectural mismatch: starting from components satisfying a property $\mathcal{P}$, the designer constructs a system that does not satisfy $\mathcal{P}$. As another example, we mention architectural mismatches arising in case of underspecification: starting from components some of which have a behavior that partially depends on the interactions with other components, the designer ends up with a system whose behavior is not fully specified. In this paper we concentrate on deadlock related architectural mismatches.

There are many causes of architectural mismatches that lead to a system block. In this paper we examine the following three causes: (1) incompatibility between two components due to a single interaction, (2) incompatibility between two components due to the combination of several interactions, and (3) lack of interoperability among a set of components forming a cyclic topology. The contribution of this paper is to develop a suitable formal framework in which all the three causes above of deadlock related architectural mismatches are dealt with through standard analysis techniques. Since component based software systems are considered, we develop an ADL called PADL that allows compositional specifications to be written through a process algebra and provides the designer with powerful means like observational equivalences [Milner 1989].

Cause (1) is the simplest example of deadlock related architectural mismatch: for instance, a deadlock free component that wants to send data to another deadlock free component not willing to accept data from the first component. This kind of architectural mismatch is considered in the process algebraic framework of Wright [Allen and Garlan 1997]. The designer using Wright is required to specify the ports of each component and the roles of each component connector. A port represents the behavior of a component w.r.t. an interaction through a connector, while the roles of a connector describe the expected interactions of each component attached to the connector. Given a connector and the set of components attached to it, all the pairs of corresponding ports and roles are verified to be compatible, which ensures a deadlock free interaction of the set of components through that connector provided that a constraint on the overall interaction of the roles of the connector is satisfied. In [Inverardi et al. 2000] it is argued that the presence of deadlock related architectural mismatches should not be checked at the port/role level only but also at the component level and should be carried out efficiently. The reason is that the internal (as opposed to interactional or observable) behavior of a component may correlate different interactions of the component itself in such a way that the system blocks (cause (2)). As an example, in [Inverardi et al. 2000] a compressing proxy system is considered, whose deadlock free gzip component interacts properly with the rest of the system as long as its input flow interaction and its output flow interaction are examined separately. However, since the gzip component can autonomously decide to start sending compressed data before having received all the data to be compressed, a deadlock is detected when examining the two flows together w.r.t. the rest of the system. This kind of architectural mismatch is dealt with in [Inverardi et al. 2000; Inverardi and Uchitel 2001] by extracting from each component description its actual behavior and its assumptions about the behavior of the rest of the system. An algorithm based on a variant of an observational equivalence is then defined that tries to match each assumed behavior with a sequence of actual behaviors. If the algorithm succeeds, the set of interacting components is deadlock free. In PADL we treat cause (1) and (2) by means of an architectural compatibility check based on a standard observational equivalence. Given the set of components interacting through a given deadlock free component K of a software system, the architectural compatibility check ensures that the overall interaction of those components is deadlock free provided that, for each component interacting through K, a certain observational equivalence based condition involving only the component and K is satisfied. Additionally, the architectural compatibility check scales to the whole architecture in case of acyclic topology. This architectural compatibility check, which subsumes the compatibility check of [Allen and Garlan 1997] and equals the effectiveness of the algorithm of [Inverardi et al. 2000; Inverardi and Uchitel 2001], thus bringing different techniques into a uniform framework using standard machinery, is both efficient, as it avoids the construction of the state space of the overall interaction, and helpful in case of mismatch detection, as it allows the source of the mismatch to be singled out.

As far as cause (3) is concerned, we observe that the deadlock related architectural mismatches should not be checked only between pairs of interacting components but also among sets of interacting components that form a cycle. The reason is that in such a scenario each pair composed of two interacting components could be well formed, while the global interaction might not be. In terms of the algorithm of [Inverardi et al. 2000; Inverardi and Uchitel 2001], the interactions within the sequence of actual behaviors matching an assumed behavior should be taken into account as well. As an example of this kind of mismatch, let us elaborate on the guest analogy of [Inverardi et al. 2000]. Here we have three deadlock free components: the guest, the host, and the waiter. When arriving at the party, the guest expects to be welcome by the host and to be asked whether (s)he wants an orange juice or a pineapple juice. The guest then expects that the host tells the waiter what the desired drink is and that the waiter brings that drink. Suppose that the interactions within the pairs of components are correct, but assume that the host has bad memory or is a malicious person and can thus tell the waiter to bring to the guest a drink different from the desired one. It is clear that the three components do not interoperate correctly when considered collectively. PADL comes equipped with an architectural interoperability check that (like the architectural compatibility check for acyclic topologies) is based on a standard observational equivalence. The architectural interoperability check guarantees the absence of deadlock for sets of interacting components that form a cyclic topology provided that there is at least one deadlock free component in the cycle that satisfies a certain observational equivalence based condition (different from the one for the architectural compatibility check). An important property of the architectural interoperability check is that, in case of mismatch detection, the temporal logic based diagnostic information associated with an equivalence checking failure can be exploited during the application of a procedure to identify the source of the mismatch in the cycle.

Finally, we address the issue of scaling the architectural compatibility and interoperability checks to architectural styles through an extension of PADL. Architectural styles [Shaw and Garlan 1996] are organizational patterns that have been developed over the years as designers recognized the value of specific organizational principles and structures for certain classes of software. The aim is to capitalize on codified principles and experience to specify, analyze, plan, and monitor the construction of complex software systems with high levels of efficiency and confidence. As examples of architectural styles we have client-server systems, pipe-filter organizations, layered architectures, and so on. An architectural style defines a family of software systems having a common vocabulary of components as well as a common topology and set of contraints on the interactions among the components. Since an architectural style encompasses an entire family of software systems, it is desirable to formalize the concept of architectural style both to have a precise definition of the system family and to study the architectural properties common to all the systems of the family. The formalization of an architectural style is complicated by the presence of two degrees of freedom within the set of instances of the style: variability of the internal behavior of the components and variability of the topology formed by the components. As a first step towards the solution of the problem, we propose an intermediate abstraction called architectural type, whose instances differ only for the internal behavior of their components, while sharing the same topology. We define an efficient architectural conformity check based on a standard obervational equivalence to verify whether an architecture is an instance of an architectural type. We show that all the architectures conforming to the same architectural type possess the same compatibility and interoperability properties.

This paper, which is an extended and revised version of [Bernardo et al. 2000; 2001], is organized as follows. In Sect. 2 we recall some notions about process algebras and observational equivalences that are used in the rest of the paper. In Sect. 3 we present the textual and graphical notation for PADL as well as its semantics defined by translation into process algebra. In Sect. 4 we introduce the architectural compatibility check and we assess its adequacy on a compressing proxy system. In Sect. 5 we describe the architectural interoperability check and we assess its adequacy on a cruise control system. In Sect. 6 we propose the concept of architectural type, we provide the architectural conformity check, and we demonstrate the scalability of the architectural compatibility and interoperability checks. Finally, in Sect. 7 we report some concluding remarks about future work, while comparisons with related work are performed in every section.

2. PROCESS ALGEBRAS

Process algebras [Milner 1989; Hoare 1985] are algebraic languages that support the compositional description of concurrent and distributed systems and the formal verification of their properties. In this section we recall the basic notions about syntax, semantics, and observational equivalences for process algebras that will be used in the rest of the paper.

2.1 Syntax and Operational Semantics

The basic elements of any process algebra are its actions, which represent activities carried out by the systems being modeled, and its operators (among which a parallel composition operator), which are used to compose process algebraic descriptions.

Definition 2.1. The set of process terms of the process algebra PA [1] we consider in this paper is generated by the following syntax

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

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 $ARFun$ of action relabeling functions preserving observability (i.e., $\varphi^{-1}(\tau) = \{\tau\}$), and A belongs to a set $Const$ of constants each possessing a (possibly recursive) defining equation of the form $A \stackrel{\Delta}{=} E$. ■

[1] A mix of CCS [Milner 1989] and CSP [Hoare 1985], which is the functional kernel of the stochastic process algebra EMPA$_{\mathrm{gr}}$ [Bravetti and Bernardo 2000].

Table I. Operational semantics for PA

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

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

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

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

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

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

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

In the syntax above, the null term "$\underline{0}$" is the term that cannot execute any action. The action prefix operator "$a.\_$" denotes the sequential composition of an action and a term: term $a.E$ can execute action a and then behaves as term E. The hiding operator "$\_/L$" makes some of the executed actions unobservable: term E/L behaves as term E with each executed action a turned into τ whenever $a \in L$. The relabeling operator "$\_[\varphi]$" changes the executed actions: term $E[\varphi]$ behaves as term E with each executed action a turned into $\varphi(a)$. The alternative composition operator "$\_+\_$" expresses a nondeterministic choice between two terms: term $E_1 + E_2$ behaves as either term E_1 or term E_2 depending on whether an action of E_1 or an action of E_2 is executed. The parallel composition operator "$\_ \parallel_S \_$" expresses the concurrent execution of two terms according to the following synchronization discipline: two (observable) actions can synchronize iff they belong to the synchronization set S and are equal. Term $E_1 \parallel_S E_2$ asynchronously executes actions of E_1 or E_2 not belonging to S and synchronously executes actions of E_1 and E_2 meeting the requirement above. 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 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 is a state transition graph, where states are in correspondence with process terms and transitions are labeled with actions. In order to get as usual finitely branching state transition graphs, 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. The operational semantics for the closed and guarded terms of PA is shown in Table I. As an example, the first rule for the hiding operator means that, if the current state is E/L and E is capable of performing an action $a \notin L$ thus evolving into E', then E/L is capable of performing the same action a thus evolving into E'/L. The other rules must be read in the same way. In particular, note that in the third rule for the parallel composition operator the action resulting from the synchronization of two a actions is still a. This means that multiway synchronizations are permitted.

2.2 Observational Equivalences

Due to their algebraic nature, process description languages like PA naturally lend themselves to the definition of equivalences and preorders. In particular, in the concurrency theory literature several notions of equivalence can be found, which relate terms according to a certain interpretation of their behavior (see [van Glabbeek 2001] for a survey).

Here we recall the weak bisimulation equivalence [Milner 1989], which captures the ability of two terms to simulate each other's behaviors up to τ actions, i.e. when abstracting from internal details. Before introducing it, we generalize the transition relation $\longrightarrow$ labeled with actions to the transition relation $\Longrightarrow$ labeled with sequences of actions

$$\xRightarrow{a_1 \dots a_n} \; \equiv \; \xrightarrow{a_1} \dots \xrightarrow{a_n}$$

with ε denoting the empty sequence and $\xRightarrow{\varepsilon}$ being the identity relation over process terms. Moreover, if σ is a sequence over Act, let $\hat{\sigma}$ denote the sequence over $Act - \{\tau\}$ obtained from σ by removing all the occurring τ actions. Finally, we use

$$\overset{\sigma}{\models\!\!\!\Longrightarrow} \; \equiv \; \xRightarrow{\tau^m} \xRightarrow{\sigma} \xRightarrow{\tau^n}$$

with $m, n \in \mathbb{N}$, to indicate the execution of an action sequence σ possibly preceded and followed by the execution of arbitrarily many unobservable actions.

Definition 2.2. A binary relation $\mathcal{B}$ over PA is a weak bisimulation iff, whenever $(E_1, E_2) \in \mathcal{B}$, then for all $a \in Act$: [2]

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

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

The union of all the weak bisimulations, denoted by $\approx_{\mathrm{B}}$, is called the weak bisimulation equivalence. ■

$\approx_{\mathrm{B}}$ enjoys several algebraic properties w.r.t. the dynamic operators. Although such properties are not explicitly used in the rest of the paper, their are reported below to show the ability of $\approx_{\mathrm{B}}$ to abstract from τ actions, which makes $\approx_{\mathrm{B}}$ well suited to reason on projections (obtained through the hiding operator) of the behavior of

[2] Note that $\hat{a} = a$ for $a \neq \tau$, $\hat{a} = \varepsilon$ for $a = \tau$.

system components:

$$
\begin{aligned}
(E_1 + E_2) + E_3 &\approx_{\mathrm{B}} E_1 + (E_2 + E_3) \\
E_1 + E_2 &\approx_{\mathrm{B}} E_2 + E_1 \\
E + \underline{0} &\approx_{\mathrm{B}} E \\
E + E &\approx_{\mathrm{B}} E \\
\tau.E &\approx_{\mathrm{B}} E \\
a.\tau.E &\approx_{\mathrm{B}} a.E \\
E + \tau.E &\approx_{\mathrm{B}} \tau.E \\
a.(E_1 + \tau.E_2) + a.E_2 &\approx_{\mathrm{B}} a.(E_1 + \tau.E_2)
\end{aligned}
$$

Furthermore, $\approx_{\mathrm{B}}$ is a congruence w.r.t. the static operators. This means that the observable behavior of a term does not change if we replace a subterm with a weakly bisimulation equivalent term in the context of a static operator. Formally, if $E_1 \approx_{\mathrm{B}} E_2$ then

$$
\begin{aligned}
E_1/L &\approx_{\mathrm{B}} E_2/L \\
E_1[\varphi] &\approx_{\mathrm{B}} E_2[\varphi] \\
E_1 \,\|_S\, E &\approx_{\mathrm{B}} E_2 \,\|_S\, E
\end{aligned}
$$

This property, together with the fact that $\approx_{\mathrm{B}}$ never equates a deadlocked term to a deadlock free term, is essential to prove the deadlock freedom results contained in this paper.

Definition 2.3. A term E is said to be deadlock free iff for each state s of its underlying state transition graph there exist an observable action a and a state s' such that $s \overset{a}{\Longmapsto} s'$. ■

3. A PROCESS ALGEBRA BASED ADL

In this section we present the syntax and the semantics for PADL, an ADL based on PA for the compositional and hierarchical modeling of architectural types. In this section we concentrate on the description of a single software architecture, deferring to Sect. 6 the detailed explanation of the concept of architectural type and the related notion of conformity of a software architecture to an architectural type. Throughout this section, we consider a pipe-filter system to exemplify the features of PADL.

3.1 Textual Notation

A description in PADL represents an architectural type, which we view as a single software architecture instead of a family of software architectures for the time being. As shown in Table II, 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 as a family of sequential PA terms, 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 software 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. Every interaction is declared to be an input interaction or an output interaction and the attachments must respect such a classification: every attachment must

Table II. Structure of a PADL textual description

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	

involve an output interaction and an input interaction of two different AEIs. An attachment between two interactions of the same AEI is not allowed because the interactions of an AEI are logically intended for communications with other AEIs; if needed, such an attachment can be simulated within the behavior of the AEI. An AEI can have different types of interactions (input/output, local/architectural); it must have at least one local interaction. Every local interaction must be involved in at least one attachment, hence there cannot be any isolated AEI, while every architectural interaction must not be involved in any attachment. In order to allow several AEIs to synchronize, every local interaction can be involved in several attachments provided that no autosynchronization arises, i.e. no chain of attachments is created that starts from a local interaction of an AEI and terminates on a local interaction of the same AEI.

We now illustrate each part of an architectural description in PADL by means of an example concerning a pipe-filter system.[3] The system is composed of three 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 two downstream filters according to the availability of free positions in their buffers. If both have free positions, the choice is resolved nondeterministically.

The first part of a PADL description defines the name of the architectural type:

archi_type *PipeFilter*

The specification above indicates that the architectural type for the pipe-filter system is named *PipeFilter*.

The second part of the description defines the AETs on which the architectural type is based. Every AET is defined through its name, its behavior, which is formalized by a family of sequential PA terms, and its interactions, which are given by a set of PA actions occurring in the behavior. The interactions represent a means whereby a generic instance of the AET communicates and cooperates with the rest of the system. Every interaction is declared to be an input interaction or an output interaction:

[3] PADL keywords will be written in boldface.

$$
\begin{array}{l}
\textbf{archi\_elem\_types} \\
\quad \textbf{elem\_type}\ \mathit{FilterT} \\
\qquad \textbf{behavior}\ \mathit{Filter} \triangleq \mathit{accept\_item}.\mathit{Filter}' + \\
\qquad\qquad\qquad\qquad \mathit{fail}.\mathit{repair}.\mathit{Filter} \\
\qquad\qquad\qquad \mathit{Filter}' \triangleq \mathit{accept\_item}.\mathit{Filter}'' + \\
\qquad\qquad\qquad\qquad \mathit{serve\_item}.\mathit{Filter} + \\
\qquad\qquad\qquad\qquad \mathit{fail}.\mathit{repair}.\mathit{Filter}' \\
\qquad\qquad\qquad \mathit{Filter}'' \triangleq \mathit{serve\_item}.\mathit{Filter}' + \\
\qquad\qquad\qquad\qquad \mathit{fail}.\mathit{repair}.\mathit{Filter}'' \\
\qquad \textbf{interactions}\ \textbf{input}\ \mathit{accept\_item} \\
\qquad\qquad\qquad \textbf{output}\ \mathit{serve\_item} \\
\quad \textbf{elem\_type}\ \mathit{PipeT} \\
\qquad \textbf{behavior}\ \mathit{Pipe} \triangleq \mathit{accept\_item}.(\mathit{forward\_item}_1.\mathit{Pipe} + \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{forward\_item}_2.\mathit{Pipe}) \\
\qquad \textbf{interactions}\ \textbf{input}\ \mathit{accept\_item} \\
\qquad\qquad\qquad \textbf{output}\ \mathit{forward\_item}_1, \mathit{forward\_item}_2
\end{array}
$$

The specification above indicates that AET *FilterT* has the behavior described by term *Filter*. This represents a service center of capacity two that can fail and be repaired, which interacts with the rest of the system by accepting new items and delivering served items. Initially, besides failing and being repaired, the filter can only accept an item coming from the outside, thus evolving into Filter'. If an item is in the buffer (Filter') and the filter does not fail, then either another item from the outside is accepted or the item in the buffer is served. If two items are in the buffer (Filter'') and the filter does not fail, no more items can be accepted; the filter can only serve one of the waiting items. AET *PipeT*, instead, has the behavior described by term *Pipe*, which repeatedly accepts an item and forwards it along one of two different routes. Accepting new items and forwarding them are the only interactions with the rest of the system.

The third part of the description defines the topology of the architectural type through the declaration of the instances of the previously introduced AETs, some of the interactions of such AEIs as being architectural, and the attachments among the interactions of such AEIs. The AEIs represent an abstraction of the software components constituting the system. The architectural interactions are interactions of some of the AEIs that act as interfaces for the whole architectural type. Their role is related to hierarchical modeling, as will be made clear in Sect. 6 when explaining the concept of architectural type. Every attachment connects an output interaction of an AEI to an input interaction of another AEI. In order to avoid ambiguity, every interaction declared as being architectural or involved in an attachment is expressed through the dot notation, with its name prefixed by the name of the AEI to which it belongs. Every local interaction must be involved in at least one attachment, otherwise the related action would simply represent an internal activity of the AEI to which it belongs. On the contrary, every architectural interaction cannot be involved in any attachment, as its purpose is to serve when embedding the architectural type to which it belongs in the context of a larger architectural

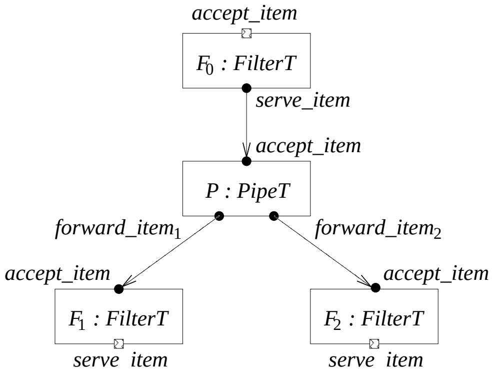

Fig. 1. Flow graph of *PipeFilter*

type as we shall see in Sect. 6. In order to allow several AEIs to synchronize, every local interaction can be involved in several attachments provided that no autosynchronization arises, i.e. no chain of attachments is created that starts from a local interaction of an AEI and terminates on a local interaction of the same AEI:

```
archi_topology
   archi_elem_instances $F_0, F_1, F_2 : \mathit{FilterT}$
                        $P : \mathit{PipeT}$
   archi_interactions input $F_0.\mathit{accept\_item}$
                      output $F_1.\mathit{serve\_item}, F_2.\mathit{serve\_item}$
   archi_attachments from $F_0.\mathit{serve\_item}$ to $P.\mathit{accept\_item}$
                     from $P.\mathit{forward\_item}_1$ to $F_1.\mathit{accept\_item}$
                     from $P.\mathit{forward\_item}_2$ to $F_2.\mathit{accept\_item}$
```

The specification above establishes that there are three instances F_0, F_1, and F_2 of *FilterT* as well as one instance P of *PipeT*, connected in such a way that the items flow from F_0 to P and from P to F_1 or F_2. Additionally, the *accept_item* interaction of F_0 and the *serve_item* interactions of F_1 and F_2 are declared as being architectural. We shall see in Sect. 6 how they are used in the description of a client-server system where the server structure is given by the pipe-filter organization above.

3.2 Graphical Notation

PADL comes equipped with a graphical notation to provide a visual help during the development of the architecture of complex software systems. Such a graphical notation is based on flow graphs [Milner 1989]. In a flow graph representing an architectural description in PADL, the boxes denote the AEIs, the black circles denote the local interactions, the white squares denote the architectural interactions, and the directed edges denote the attachments. Due to the absence of isolated AEIs, every considered flow graph will always be connected. As an example, the architectural type *PipeFilter* can be pictorially represented through the flow graph of Fig. 1.

From a methodological viewpoint, when modeling an architectural type with PADL, it is convenient to start with the flow graph representation of the architectural type and then to textually specify the behavior of each AET.

3.3 Translation Semantics into PA

The semantics of a PADL specification is given by translation into PA. While only the dynamic operators of PA can be used in the syntax of a PADL specification, the more complicated static operators of PA are transparently used in the semantics of a PADL specification. The translation into PA is accomplished in two steps. In the first step, the semantics of all the instances of each AET is defined to be the behavior of the 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 instances of the AET.

Definition 3.1. Given a PADL specification, let $\mathcal{C}$ be an AET with behavior E and interaction set $\mathcal{I}$. The semantics of $\mathcal{C}$ and its instances is defined by $[\![\mathcal{C}]\!] = E/(Act - \{\tau\} - \mathcal{I})$. ■

In our pipe-filter example we have

$$\begin{array}{rcccl} [\![FilterT]\!] & = & [\![F_0]\!] = [\![F_1]\!] = [\![F_2]\!] & = & Filter/\{fail, repair\} \\ [\![PipeT]\!] & = & [\![P]\!] & = & Pipe \end{array}$$

thus abstracting from the internal activities *fail* and *repair*. The state transition graph underlying $[\![FilterT]\!]$ has 6 states and 10 transitions: among such transitions, 4 are observable while 6 are invisible (they represent the execution of *fail* and *repair* actions). The state transition graph underlying $[\![PipeT]\!]$, instead, has 2 states and 3 transitions.

In the second step, the semantics of an architectural type is obtained by composing in parallel the semantics of its AEIs according to the specified attachments. Recalled that the parallel composition operator is left associative, in our pipe-filter example we have

$$\begin{array}{rcl} [\![PipeFilter]\!] & = & [\![F_0]\!][serve\_item \mapsto a] \parallel_{\emptyset} \\ & & \quad [\![F_1]\!][accept\_item \mapsto a_1] \parallel_{\emptyset} \\ & & \qquad [\![F_2]\!][accept\_item \mapsto a_2] \parallel_{\{a,a_1,a_2\}} \\ & & \qquad\quad [\![P]\!][accept\_item \mapsto a, \\ & & \qquad\qquad forward\_item_1 \mapsto a_1, \\ & & \qquad\qquad forward\_item_2 \mapsto a_2] \end{array}$$

The use of the relabeling operator is necessary to make the AEIs interact. As an example, F_0 and P must interact via *serve_item* and *accept_item*, which are different from each other. Since the parallel composition operator allows only equal actions to synchronize, in $[\![PipeFilter]\!]$ each *serve_item* action executed by $[\![F_0]\!]$ and each *accept_item* action executed by $[\![P]\!]$ 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 $[\![F_0]\!]$ nor in $[\![P]\!]$. Then a synchronization on a is forced between the relabeled versions of $[\![F_0]\!]$ and $[\![P]\!]$ by means of operator $\parallel_{\{a,a_1,a_2\}}$. The state transition graph underlying $[\![PipeFilter]\!]$ has 432 states and 1944 transitions: among such transitions, 648 are observable while 1296 are invisible (they represent the execution of *fail* and *repair* actions by F_0, F_1, and F_2).

In order to define the semantics of an arbitrary architectural type, first 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 chains of attachments, as each of them corresponds to a maximal set of synchronizing interactions, all of which must therefore be relabeled to the same fresh action. Given an architectural type $\mathcal{A}$, let $C_1, \dots, C_n$ be some of its AEIs and let i, j, k range over $\{1, \dots, n\}$. For each AEI C_i, let $\mathcal{I}_{C_i}$ be the set of its interactions, $\mathcal{AI}_{C_i} \subseteq \mathcal{I}_{C_i}$ be the set of its interactions declared as being architectural, and $\mathcal{LI}_{C_i;C_1,\dots,C_n} \subseteq \mathcal{I}_{C_i} - \mathcal{AI}_{C_i}$ be the set of its local interactions attached to local interactions of $C_1, \dots, C_n$. We now formalize the concept of chain of attachments through the notion of connected set of interactions.

Definition 3.2. We say that a set $\mathcal{LI}$ of local interactions of $C_1, \dots, C_n$ is connected iff:

—for each pair $(C_i.a_1, C_j.a_2)$ of interactions of $\mathcal{LI}$, either there is an attachment between them, or there exists an interaction $C_k.a_3$ of $\mathcal{LI}$ such that there is an attachment between $C_i.a_1$ and $C_k.a_3$ and $C_k.a_3$ is connected to $C_j.a_2$;

—$\mathcal{LI}$ is maximal. ■

As an example, in *PipeFilter* there are three connected sets of local interactions:

$$\{F_0.serve\_item, P.accept\_item\}$$
$$\{P.forward\_item_1, F_1.accept\_item\}$$
$$\{P.forward\_item_2, F_2.accept\_item\}$$

Once we have identified the connected sets of local interactions, we construct a set $\mathcal{S}(C_1, \dots, C_n)$ composed of as many fresh actions as there are connected sets of local interactions. Then we relabel all the local interactions in the same connected set to the same fresh action. This is achieved by defining a set of injective action relabeling functions of the form $\varphi_{C_i;C_1,\dots,C_n} : \mathcal{LI}_{C_i;C_1,\dots,C_n} \longrightarrow \mathcal{S}(C_1, \dots, C_n)$ in such a way that $\varphi_{C_i;C_1,\dots,C_n}(a_1) = \varphi_{C_j;C_1,\dots,C_n}(a_2)$ iff $C_i.a_1$ and $C_j.a_2$ belong to the same connected 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 $C_1, \dots, C_n$. The closed semantics will be used in the definition of the architectural checks. It abstracts from the architectural interactions of C_i as these must not come into play when checking for deadlock related architectural mismatches. 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 3.3. The closed and the open interacting semantics of C_i restricted to $C_1, \dots, C_n$ are defined by

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

If we compare Def. 3.1 and Def. 3.3, we observe that the latter gives rise to a further projection on the local interactions attached to local interactions of $C_1, \dots, C_n$ and relabel such local interactions in order to make it possible the synchronization among $C_1, \dots, C_n$. 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:

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

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

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

■

Definition 3.5. The semantics of an architectural type $\mathcal{A}$ with AEIs $C_1, \ldots, C_n$ is defined by $[\![\mathcal{A}]\!] = [\![C_1, \ldots, C_n]\!]^{\mathrm{o}}$. ■

When $[\![C_1, \ldots, C_n]\!]^{\mathrm{o}}$ is the semantics of a whole architectural type, the application of the hiding operator occurring in Def. 3.3 is redundant.

We conclude by observing that the semantics of a set of AEIs, hence the semantics of an architectural type, is well defined because we do not permit AEI autosynchronizations. Additionally, we note that changing the order of the AEIs results in a change of the synchronization sets, with no modification of the underlying state transition graph. Furthermore, we point out that the mathematical details related to the definition of the translation semantics for PADL can be made completely transparent to the PADL users.

3.4 Related Work

PADL is clearly inspired by Wright [Allen and Garlan 1997]. In both languages, the behavior of the software components is specified through sequential process terms, and the semantics of the architectural descriptions is given by translation into a process term using the static operators.

However, there are several differences between PADL and Wright. First, Wright distinguishes between components and connectors, while in PADL there are just architectural elements. Each of them can be interpreted as being a component or a connector depending on the particular software 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 PADL, similarly to Darwin [Magee et al. 1995], the interactions are simply expressed as actions. Since ports and roles can be retrieved via projections (see Def. 3.3), this avoids redundancy in the specifications. Third, as will become clear in Sect. 6, PADL supports the hierarchical modeling of families of architectures, while Wright only supports the flat modeling of a single architecture. Fourth, similarly to Darwin and UniCon [Shaw et al. 1995], PADL is equipped with a graphical notation that provides a visual help to the software architect. [4] Fifth, Wright allows the connector behavior to be extended with trace predicates that further constrain the behavior itself, while this is not possible in PADL. As a consequence, in PADL the software architect is forced to completely describe each behavior in a process algebraic way and then to verify (e.g. via model checking [Clarke et al. 1999]) that certain local properties are satisfied by the behavior. This realizes a clear separation of concerns between modeling and analysis.

4. ARCHITECTURAL COMPATIBILITY CHECK

From the software architect perspective, using PADL instead of PA provides the capability of modeling complex software systems in an easier way, which in particular elucidates the basic architectural concepts while hiding some process algebra theory aspects. Since the semantics for PADL is given by translation into PA, on the analysis side the software architect may reuse the standard verification techniques for process algebras, i.e. equivalence/preorder checking and model checking. However, the software architect needs not only an easier way of modeling, but also a more controlled way of modeling that supports the automatical detection of architectural mismatches as well as the identification of the software components that cause them.

As seen in Sect. 1, in this paper we focus on three causes of deadlock related architectural mismatches. To this purpose, we enrich PADL with a set of architectural checks that take care of verifying whether the deadlock free components of a software architecture fit together well, i.e. do not lead to system blocks. In this section, we concentrate on the well formedness of acyclic architectural types, while in Sect. 5 we address the well formedness of cyclic architectural types.

Definition 4.1. Given an architectural type $\mathcal{A}$, the reduced flow graph of $\mathcal{A}$ is an indirect flow graph obtained from the flow graph of $\mathcal{A}$ by collapsing all the edges between two boxes into a single edge. $\mathcal{A}$ is said to be acyclic iff so is its reduced flow graph. ■

The reason why we consider the reduced flow graph for an architectural type is that the architectural checks that we shall define are not sensitive to the direction of the information flow or to the presence of several attachments between two AEIs rather than a single attachment. This preludes to a common handling of causes (1) and (2) outlined in Sect. 1.

4.1 Compatibility Condition and Deadlock Freedom Result

Our objective is to detect the presence of deadlock related architectural mismatches in an acyclic architectural type starting from an analysis of the local interactions of its AEIs. Given an acyclic architectural type, the question arises as to what the minimal group of AEIs to be considered is. If we take an AEI K and we consider all the AEIs $C_1, \dots, C_n$ attached to it, we can observe that they form a star topology

[4] Of course, a similar visualization facility could easily be added to Wright.

whose center is K, as the absence of cycles prevents any two AEIs among $C_1, \ldots, C_n$ from communicating via an AEI different from K. It can easily be recognized that an acyclic architectural type is just a composition of star topologies. In this section we show that the presence of deadlock related architectural mismatches due to causes (1) and (2) can be investigated at the level of the constituent star topologies through a suitable architectural compatibility check. Moreover, we show that if the architectural compatibility check reveals that all the star topologies in an acyclic architectural type are deadlock free, then the absence of deadlock scales to the whole architectural type.

Before introducing the architectural compatibility check, let us formalize the semantics of a star topology through a reworking of Def. 3.4. This takes into account the fact that the considered set of AEIs interact through a given AEI only, which simplifies the synchronization sets.

Definition 4.2. Given an acyclic architectural type, let $C_1, \ldots, C_n$ be the AEIs attached to AEI K. The interacting semantics of $C_1, \ldots, C_n$ through K is defined by

$$\begin{aligned} [\![K; C_1, \ldots, C_n]\!] \; = \; & [\![K]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} \parallel_{\mathcal{S}(K;K,C_1,\ldots,C_n)} \\ & [\![C_1]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} \parallel_{\mathcal{S}(K;K,C_1,\ldots,C_n)} \cdots \\ & \cdots \parallel_{\mathcal{S}(K;K,C_1,\ldots,C_n)} [\![C_n]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} \end{aligned} \quad \blacksquare$$

We show below that it is possible to verify that $[\![K; C_1, \ldots, C_n]\!]$ is deadlock free by locally considering the interactions of K with each C_i in turn. The idea is that C_i is compatible with K if the parallel composition of their closed interacting semantics is weakly bisimulation equivalent to the closed interacting semantics of K itself. Intuitively, this means that attaching C_i to K does not alter the behavior of K, i.e. K is designed in such a way that it suitably coordinates with C_i.

Definition 4.3. Given an acyclic architectural type, let $C_1, \ldots, C_n$ be the AEIs attached to AEI K. C_i is said to be compatible with K iff

$$[\![K]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} \parallel_{\mathcal{S}(K;K,C_1,\ldots,C_n)} [\![C_i]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} \approx_{\mathrm{B}} [\![K]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} \quad \blacksquare$$

Observe that if C_i is compatible with K and vice versa, then $[\![C_i]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} \approx_{\mathrm{B}} [\![K]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n}$. However, the reverse implication does not hold. As an example, if we take K and C_i such that

$$\begin{aligned} [\![C_i]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} & \stackrel{\Delta}{=} \tau.a.[\![C_i]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} + \tau.b.[\![C_i]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} \\ [\![K]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} & \stackrel{\Delta}{=} \tau.a.[\![K]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} + \tau.b.[\![K]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} \end{aligned}$$

with equal observable actions attached to each other, then $[\![C_i]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n} \approx_{\mathrm{B}} [\![K]\!]^{\mathrm{c}}_{K,C_1,\ldots,C_n}$ but there is no compatibility. Indeed, we have that the combination of K and C_i, which are individually deadlock free, deadlocks whenever K and C_i separately perform two invisible actions leading to two observable actions not attached to each other. This simple example demonstrates that requiring K and each C_i in turn to be weakly bisimulation equivalent is not enough to ensure deadlock freedom at the level of the star topology, [5] hence we need a stronger requirement that takes into account the coordination of K with each C_i in turn. The

[5] The example actually shows that not even isomorphism is enough.

next theorem shows that compatibility provides a sufficient condition for deadlock freedom when starting from a deadlock free center K of the star topology. The subsequent corollary shows that compatibility scales from the star topologies to the whole architectural type.

LEMMA 4.4. *Given an acyclic architectural type, let $C_1, \dots, C_n$ be the AEIs attached to AEI K. For each nonempty subset $\{C'_1, \dots, C'_{n'}\}$ of $\{C_1, \dots, C_n\}$, if C'_i is compatible with K for all $i = 1, \dots, n'$ then*

$$\begin{array}{l} \llbracket K \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \;\|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \\ \quad \llbracket C'_1 \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \;\|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \cdots \\ \qquad \cdots \|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \llbracket C'_{n'} \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \;\approx_{\mathrm{B}}\; \llbracket K \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \end{array}$$

PROOF. We proceed by induction on n':

—If $n' = 1$ then the property immediately follows from the compatibility of C'_1 with K.

—Let the property hold for a certain $n' > 1$ and consider

$$\begin{array}{l} \llbracket K \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \;\|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \\ \quad \llbracket C'_1 \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \;\|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \\ \qquad \llbracket C'_2 \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \;\|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \cdots \\ \qquad\quad \cdots \|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \llbracket C'_{n'+1} \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \end{array}$$

Since C'_1 is compatible with K, it follows that

$$\llbracket K \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \|_{\mathcal{S}(K;C,C_1,\dots,C_n)} \llbracket C'_1 \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \approx_{\mathrm{B}} \llbracket K \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n}$$

From the fact that $\approx_{\mathrm{B}}$ is a congruence w.r.t. the parallel composition operator, it follows that

$$\begin{array}{l} \llbracket K \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \;\|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \\ \quad \llbracket C'_1 \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \;\|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \\ \quad\; \llbracket C'_2 \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \;\|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \cdots \\ \qquad \cdots \|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \llbracket C'_{n'+1} \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \\ \qquad\qquad \approx_{\mathrm{B}} \; \llbracket K \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \;\|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \\ \qquad\qquad\qquad \llbracket C'_2 \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \;\|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \cdots \\ \qquad\qquad\qquad\; \cdots \|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \llbracket C'_{n'+1} \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n} \end{array}$$

The result then follows by the induction hypothesis.

□

THEOREM 4.5. *Given an acyclic architectural type, let $C_1, \dots, C_n$ be the AEIs attached to AEI K. If $\llbracket K \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n}$ is deadlock free and C_i is compatible with K for all $i = 1, \dots, n$, then $\llbracket K; C_1, \dots, C_n \rrbracket$ is deadlock free.*

PROOF. Because of the compatibility hypothesis, by Lemma 4.4 we have that $\llbracket K; C_1, \dots, C_n \rrbracket \approx_{\mathrm{B}} \llbracket K \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n}$. Since $\llbracket K \rrbracket^{\mathrm{c}}_{K,C_1,\dots,C_n}$ is deadlock free and $\approx_{\mathrm{B}}$ preserves deadlock freedom, the result follows. □

COROLLARY 4.6. *Given an acyclic architectural type, if the semantics of each AET with the architectural interactions being hidden is deadlock free and every AEI is compatible with each AEI attached to it, then the architectural type is deadlock free.*

Proof. To avoid trivial cases, assume that the architectural type has at least two AEIs. We proceed by induction on the number m of edges in the reduced flow graph of the architectural type:

—If $m = 1$ then the reduced flow graph of the architectural type has two AEIs connected by some attachments. The result immediately follows by applying Thm. 4.5.

—Let the result hold for a certain $m > 1$ and suppose that the reduced flow graph of the architectural type has $m + 1$ edges. Since the architectural type ia acyclic, there exists an AEI C having attachments to a single AEI K. Since C is compatible with K by the initial hypothesis, if $C_1, \dots, C_n$ are all the other AEIs attached to K we have

$$[\![K]\!]^{\mathrm{c}}_{K,C,C_1,\dots,C_n} \,\|_{\mathcal{S}(K;K,C,C_1,\dots,C_n)}\, [\![C]\!]^{\mathrm{c}}_{K,C,C_1,\dots,C_n} \approx_{\mathrm{B}} [\![K]\!]^{\mathrm{c}}_{K,C,C_1,\dots,C_n}$$

hence $[\![K]\!]^{\mathrm{o}}_{K,C,C_1,\dots,C_n} \,\|_{\mathcal{S}(K;K,C,C_1,\dots,C_n)}\, [\![C]\!]^{\mathrm{o}}_{K,C,C_1,\dots,C_n}$ is deadlock free because so is $[\![K]\!]^{\mathrm{c}}_{K,C,C_1,\dots,C_n}$ by the initial hypothesis. Let us replace K and C with a new AEI such that its behavior is isomorphic to $[\![K]\!]^{\mathrm{o}}_{K,C,C_1,\dots,C_n} \,\|_{\mathcal{S}(K;K,C,C_1,\dots,C_n)}$ $[\![C]\!]^{\mathrm{o}}_{K,C,C_1,\dots,C_n}$ and its interactions are the relabeled local interactions of K and C (where pairs of attached relabeled interactions are counted once) plus the architectural interactions of K and C. Then we obtain an architectural type such that its semantics is isomorphic to the original one and its reduced flow graph has m edges. The result then follows by the induction hypothesis.

□

If we apply the architectural compatibility check to the pipe-filter example of Sect. 3 we have

$$\begin{array}{llll}
[\![P]\!][\mathit{accept\_item} \mapsto a, & \|_{\{a\}} \; [\![F_0]\!]/\{\mathit{accept\_item}\} & \approx_{\mathrm{B}} & [\![P]\!][\mathit{accept\_item} \mapsto a, \\
\quad \mathit{forward\_item}_1 \mapsto a_1, & \quad\;\; [\mathit{serve\_item} \mapsto a] & & \quad \mathit{forward\_item}_1 \mapsto a_1, \\
\quad \mathit{forward\_item}_2 \mapsto a_2] & & & \quad \mathit{forward\_item}_2 \mapsto a_2]
\end{array}$$

$$\begin{array}{llll}
[\![P]\!][\mathit{accept\_item} \mapsto a, & \|_{\{a_1\}} \; [\![F_1]\!]/\{\mathit{serve\_item}\} & \approx_{\mathrm{B}} & [\![P]\!][\mathit{accept\_item} \mapsto a, \\
\quad \mathit{forward\_item}_1 \mapsto a_1, & \quad\;\; [\mathit{accept\_item} \mapsto a_1] & & \quad \mathit{forward\_item}_1 \mapsto a_1, \\
\quad \mathit{forward\_item}_2 \mapsto a_2] & & & \quad \mathit{forward\_item}_2 \mapsto a_2]
\end{array}$$

$$\begin{array}{llll}
[\![P]\!][\mathit{accept\_item} \mapsto a, & \|_{\{a_2\}} \; [\![F_2]\!]/\{\mathit{serve\_item}\} & \approx_{\mathrm{B}} & [\![P]\!][\mathit{accept\_item} \mapsto a, \\
\quad \mathit{forward\_item}_1 \mapsto a_1, & \quad\;\; [\mathit{accept\_item} \mapsto a_2] & & \quad \mathit{forward\_item}_1 \mapsto a_1, \\
\quad \mathit{forward\_item}_2 \mapsto a_2] & & & \quad \mathit{forward\_item}_2 \mapsto a_2]
\end{array}$$

Since $[\![P]\!][\mathit{accept\_item} \mapsto a, \mathit{forward\_item}_1 \mapsto a_1, \mathit{forward\_item}_2 \mapsto a_2]$ is deadlock free, we can conclude that so is $[\![\mathit{PipeFilter}]\!]$. Checking deadlock freedom directly on $[\![\mathit{PipeFilter}]\!]$ would have required the generation of hundreds of states and thousands of transitions, whereas checking architectural compatibility as done above has only required the generation of tens of states and transitions.

From a methodological viewpoint, given an acyclic architectural type the architectural compatibility check must be applied as follows. First, we verify that the semantics of each AET with the architectural interactions being hidden is deadlock free. Note that the hiding operator must be applied in order to make the architectural interactions unobservable. The reason is that we are interested in the way the AEIs coordinate with each other, but the architectural interactions cannot be involved in any attachment, so they must not come into play. Should not they be hidden, it may be the case that an AEI is considered to be deadlock free because it is always ready to execute an architectural interaction, even though it is not able to properly coordinate with the other AEIs attached to it. Then, we assess the validity of the compatibility condition for every pair of attached AEIs. We observe that Cor. 4.6 only provides a sufficient condition for deadlock freedom. Therefore, a violation of the compatibility condition for a pair of attached AEIs does not necessarily imply that the architectural type can deadlock. We point out that the compatibility condition and the probability for it to be violated strongly depend on the adopted notion of observable equivalence. Here we use the weak bisimulation equivalence, which is known to be very sensitive to the branching points in the AEI behavior. It might be the case that $\llbracket K \rrbracket^{\mathrm{c}}_{K,C_1,\ldots,C_n} \parallel_{\mathcal{S}(K;K,C_1,\ldots,C_n)} \llbracket C_i \rrbracket^{\mathrm{c}}_{K,C_1,\ldots,C_n}$ is not weakly bisimulation equivalent to $\llbracket K \rrbracket^{\mathrm{c}}_{K,C_1,\ldots,C_n}$ because, although they produce the same execution traces of observable actions, they differ for the structure of their branching points. Unfortunately, trace equivalence cannot be used in place of weak bisimulation equivalence as the former does not preserve deadlock freedom, i.e. it can equate a deadlock free term to a deadlocked one. In any case, the essence of the compatibility condition is to investigate whether some attached AEIs coordinate in an acyclic topology. Thus, independently of the specific observational equivalence adopted in the compatibility condition, we believe that it is wise for the architect to interpret a violation of the condition as a warning of the possible presence of some kind of architectural mismatch. This should lead the architect to a careful analysis of the interactions between the two attached AEIs giving rise to the violation.

Given a star topology composed of $C_1, \ldots, C_n$ attached to K, applying the architectural compatibility check is more convenient than directly checking $\llbracket K; C_1, \ldots, C_n \rrbracket$ for deadlock freedom. The first reason is that the application of the architectural compatibility check requires the construction of n state spaces each resulting from the parallel composition of only two sequential terms (see Def. 4.3), whereas the direct check requires the construction of one state space resulting from the parallel composition of $n + 1$ sequential terms (see Def. 4.2). Thus, the state space construction related complexity is $O(n)$ in the case of the architectural compatibility check, $O(\alpha^n)$ in the case of the direct check. The second reason is that, in case of deadlock, the direct check provides the architect with no diagnostic information about the deadlock source. On the contrary, the architectural compatibility check pinpoints potential sources of deadlock. Whenever the compatibility condition is violated by a pair of attached AEIs, the architect knows that the two AEIs under consideration may cause an architectural mismatch that leads to deadlock.

4.2 Related Work

Our architectural compatibility check is similar in spirit to the compatibility check presented in [Allen and Garlan 1997]. However, there are some relevant differences. First, the compatibility condition of [Allen and Garlan 1997] is based on an observational preorder given by a suitable reworking of the failure semantic based preorder of [Hoare 1985], while our compatibility condition relies on the standard weak bisimulation equivalence. Second, the compatibility condition of [Allen and Garlan 1997] imposes a global constraint on the overall interaction of the roles of a connector, while our compatibility condition can be checked more efficiently as it only imposes local constraints on the interactions between pairs of attached AEIs. Third, the compatibility condition of [Allen and Garlan 1997] deals with single component ports and connector roles without taking into account the relationships among different ports of a component attached to different roles of the same connector, thus forcing the architect to merge the ports above into a single one at the modeling level in order for the compatibility check to be effective. Our compatibility condition, instead, naturally deals with both causes (1) and (2) of deadlock related architectural mismatches, as the whole behavior of the AEIs is considered. Finally, our architectural compatibility check scales to the whole architectural type in case of acyclic topology.

Our architectural compatibility check also has some similarities with the algorithm presented in [Inverardi et al. 2000; Inverardi and Uchitel 2001]. The idea of the algorithm is that the presence of mismatches should be checked not only at the port/role level but also at the component level, and that such a check should be carried out in an efficient way. For this reason, the algorithm extracts from each component description its actual behavior and its assumptions about the behavior of the rest of the system. Then, the algorithm tries to match each assumed behavior with a sequence of actual behaviors using a variant of the weak bisimulation equivalence. Our architectural compatibility check has the same objectives as the algorithm above, but relies on the standard weak bisimulation equivalence.

In conclusion, we can say that our architectural compatibility check subsumes the compatibility check of [Allen and Garlan 1997] and equals the effectiveness of the algorithm of [Inverardi et al. 2000; Inverardi and Uchitel 2001], thus bringing into a uniform, process algebraic framework different techniques for detecting deadlock related architectural mismatches using standard machinery.

4.3 Example: A Compressing Proxy System

In this section we assess the validity of the architectural compatibility check to discover a known architectural mismatch in the compressing proxy system examined in [Inverardi et al. 2000].

A compressing proxy system aims at improving the performance of Unix based Web browsers over slow networks by creating a HTTP server that compresses data before sending them across the network. This is achieved by wedding the gzip compression program to the standard HTTP server. A standard HTTP server is a series of filters strung together, which communicate through a function call based stream interface that allows an upstream filter to push data into a downstream filter. The gzip program is instead a Unix filter communicating through pipes. An important difference between the gzip program and the standard HTTP filters is that the gzip program explicitly chooses when to read, while the HTTP servers are forced to read when data are pushed to them. The gzip program may attempt to output a portion of the compressed data, perhaps because an internal buffer is full, before finishing reading all the input data.

In order to assemble the compressing proxy system from the existing HTTP server and gzip program without modifications, we must insert the gzip program into the HTTP filter series using an appropriate adaptor because of their different communication mechanisms. The flow graph of the corresponding PADL descrip-

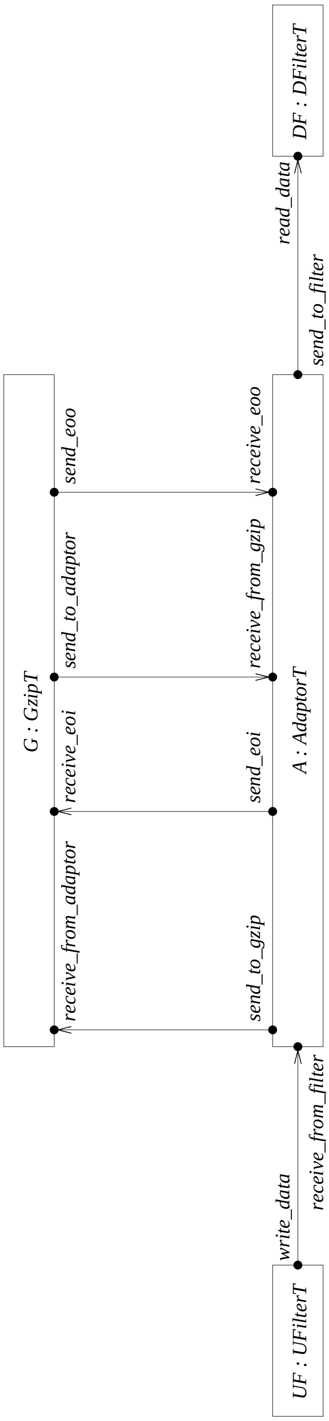

Fig. 2. Flow graph of the compressing proxy system

Table III. AET behaviors for the compressing proxy system

$\mathit{UFilter}$	$\triangleq$	$\mathit{write\_data}.\mathit{UFilter}$
$\mathit{Adaptor}$	$\triangleq$	$\mathit{receive\_from\_filter}.\mathit{Adaptor}_1$
$\mathit{Adaptor}_1$	$\triangleq$	$\mathit{send\_to\_gzip}.\mathit{Adaptor}_2$
$\mathit{Adaptor}_2$	$\triangleq$	$\mathit{send\_to\_gzip}.\mathit{Adaptor}_2 +$ $\mathit{send\_eoi}.\mathit{Adaptor}_3$
$\mathit{Adaptor}_3$	$\triangleq$	$\mathit{receive\_from\_gzip}.\mathit{Adaptor}_4$
$\mathit{Adaptor}_4$	$\triangleq$	$\mathit{receive\_from\_gzip}.\mathit{Adaptor}_4 +$ $\mathit{receive\_eoo}.\mathit{Adaptor}_5$
$\mathit{Adaptor}_5$	$\triangleq$	$\mathit{send\_to\_filter}.\mathit{Adaptor}$
Gzip	$\triangleq$	$\mathit{receive\_from\_adaptor}.\mathit{Gzip}_1$
Gzip_1	$\triangleq$	$\mathit{receive\_from\_adaptor}.\mathit{Gzip}_1 +$ $\mathit{receive\_eoi}.\mathit{Gzip}_2 +$ $\mathit{signal\_buffer\_full}.\mathit{Gzip}_2$
Gzip_2	$\triangleq$	$\mathit{send\_to\_adaptor}.\mathit{Gzip}_3$
Gzip_3	$\triangleq$	$\mathit{send\_to\_adaptor}.\mathit{Gzip}_3 +$ $\mathit{send\_eoo}.\mathit{Gzip}$
$\mathit{DFilter}$	$\triangleq$	$\mathit{read\_data}.\mathit{DFilter}$

tion is depicted in Fig. 2; its reduced version is acyclic. The behavior of each AET is shown in Table III. The underlying state transition graph has 7 states and 9 transitions (8 observable, 1 invisible).

If we apply the architectural compatibility check, we discover that

$$\begin{array}{lll}
\mathit{Adaptor}[\mathit{send\_to\_gzip} \mapsto a_1, & \|_{\{a_1,a_2,a_3,a_4\}} & \mathit{Gzip}/\{\mathit{signal\_buffer\_full}\} \\
\quad \mathit{send\_eoi} \mapsto a_2, & & \quad [\mathit{receive\_from\_adaptor} \mapsto a_1, \\
\quad \mathit{receive\_from\_gzip} \mapsto a_3, & & \quad \mathit{receive\_eoi} \mapsto a_2, \\
\quad \mathit{receive\_eoo} \mapsto a_4, & & \quad \mathit{send\_to\_adaptor} \mapsto a_3, \\
\quad \mathit{receive\_from\_filter} \mapsto a_5, & & \quad \mathit{send\_eoo} \mapsto a_4] \\
\quad \mathit{send\_to\_filter} \mapsto a_6] & & \\
& \not\approx_{\mathrm{B}} & \mathit{Adaptor}[\mathit{send\_to\_gzip} \mapsto a_1, \\
& & \quad \mathit{send\_eoi} \mapsto a_2, \\
& & \quad \mathit{receive\_from\_gzip} \mapsto a_3, \\
& & \quad \mathit{receive\_eoo} \mapsto a_4, \\
& & \quad \mathit{receive\_from\_filter} \mapsto a_5, \\
& & \quad \mathit{send\_to\_filter} \mapsto a_6]
\end{array}$$

In fact, the state transition graph of the whole architectural type has one deadlock state. A simply passes uncompressed data to G whenever it receives data from UF. Once the data stream is closed, A reads the compressed data from G and pushes them to DF. This behavior of A gives rise to an interaction with G that is not correct. More precisely, A and G fit together well as long as the uncompressed data flow from A to G and the compressed data flow from G to A are considered separately. However, when we consider the two flows together, a deadlock arises because A does not take into account the fact that G can autonomously decide to start sending compressed data back to A before having received from A all the uncompressed data.

Table IV. Correct behavior for the adaptor

$$\begin{aligned}
\mathit{Adaptor} &\triangleq \mathit{receive\_from\_filter}.\mathit{Adaptor}_1\\
\mathit{Adaptor}_1 &\triangleq \mathit{send\_to\_gzip}.\mathit{Adaptor}_2\\
\mathit{Adaptor}_2 &\triangleq \mathit{send\_to\_gzip}.\mathit{Adaptor}_2 +\\
&\ \mathit{send\_eoi}.\mathit{Adaptor}_3 +\\
&\ \mathit{signalled\_buffer\_full}.\mathit{Adaptor}_6\\
\mathit{Adaptor}_3 &\triangleq \mathit{receive\_from\_gzip}.\mathit{Adaptor}_4\\
\mathit{Adaptor}_4 &\triangleq \mathit{receive\_from\_gzip}.\mathit{Adaptor}_4 +\\
&\ \mathit{receive\_eoo}.\mathit{Adaptor}_5\\
\mathit{Adaptor}_5 &\triangleq \mathit{send\_to\_filter}.\mathit{Adaptor}\\
\mathit{Adaptor}_6 &\triangleq \mathit{receive\_from\_gzip}.\mathit{Adaptor}_7\\
\mathit{Adaptor}_7 &\triangleq \mathit{receive\_from\_gzip}.\mathit{Adaptor}_7 +\\
&\ \mathit{receive\_eoo}.\mathit{Adaptor}_1
\end{aligned}$$

To solve the problem, a new attachment must be introduced in Fig. 2 from G to A, so that G can inform A about its intention to prematurely start writing compressed data in the case that its buffer becomes full. Furthermore, A must be redesigned in such a way that it handles data incrementally, with the possibility of reading compressed data from G whenever the buffer of G becomes full. The new behavior of A is shown in Table IV, where *signalled_buffer_full* is attached to *signal_buffer_full*. By virtue of Thm. 4.5, the newly obtained acyclic architectural type is deadlock free, because the closed interacting semantics of A is deadlock free and every pair composed of A and an AEI attached to A passes the architectural compatibility check.

5. ARCHITECTURAL INTEROPERABILITY CHECK

Checking architectural compatibility is unfortunately not enough to guarantee that a software system is deadlock free in the cyclic case, as there may be further causes of architectural mismatch. Let us consider e.g. the flow graph of Fig. 3 expanded with the AET behaviors, which represents the guest analogy mentioned in Sect. 1. We recall that there are three deadlock free components: the guest, the host, and the waiter. When arriving at the party, the guest expects to be welcome by the host and to be asked whether (s)he wants an orange juice or a pineapple juice. The guest then expects that the host tells the waiter what the desired drink is and that the waiter brings that drink. Unfortunately, the host has bad memory or is a malicious person and can thus tell the waiter to bring to the guest a drink different from the desired one. Although G, H, and W are deadlock free and both G and W are compatible with H, the whole system is not deadlock free. If we construct the state transition graph underlying this architectural type, which has 7 states and 8 transitions, we discover that there are two deadlock states. Here the problem is that G and W do not communicate directly, so e.g. if G asks H for an orange juice and H tells W to bring a pineapple juice instead, G and W do not interact any more thus causing the system to block. A similar situation would happen if we had a deaf waiter instead of a strange host, who often misunderstands what the host orders and brings to the guest a drink different from the desired one. The situation

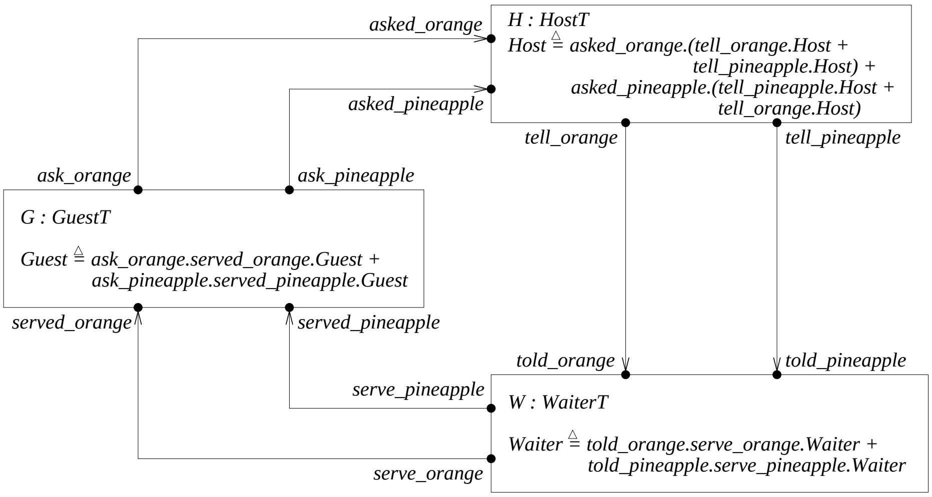
H : HostT
Host ≜ asked_orange.(tell_orange.Host + tell_pineapple.Host) + asked_pineapple.(tell_pineapple.Host + tell_orange.Host)
asked_orange
asked_pineapple
tell_orange
tell_pineapple
ask_orange
ask_pineapple
G : GuestT
Guest ≜ ask_orange.served_orange.Guest + ask_pineapple.served_pineapple.Guest
served_orange
served_pineapple
told_orange
told_pineapple
serve_pineapple
W : WaiterT
Waiter ≜ told_orange.serve_orange.Waiter + told_pineapple.serve_pineapple.Waiter
serve_orange

Fig. 3. Flow graph of the guest analogy

would be even worse with a waiter fond of fruit juices, who sometimes decides to drink the juice instead of bringing it to the guest.

If we look carefully at the flow graph of Fig. 3, we note that G, H, and W form a cycle. If we view H as being in the middle between G and W, we have that H gives rise to a mismatched interaction between G and W that cannot be detected by the architectural compatibility check. As far as the detection of deadlock related architectural mismatches is concerned, the guest analogy and its variants demonstrate that it is not always enough to consider pairs of attached AEIs. [6] The reason is that cycles of deadlock free AEIs can result in deadlocks as well, which is cause (3) identified in Sect. 1. In this section we develop another architectural check to address the well formedness of cyclic architectural types.

5.1 Interoperability Condition and Deadlock Freedom Result

Our objective is to detect the presence of deadlock related architectural mismatches in a cyclic architectural type starting from an analysis of the local interactions of its AEIs. Obviously, the minimal group of AEIs to be considered is given by a set $C_1, \dots, C_n$ of AEIs forming a cyclic topology. In this section we show that the presence of deadlock related architectural mismatches due to cause (3) can be investigated for cycles of AEIs through a suitable architectural interoperability check. More precisely, we prove that it is possible to verify that $[\![C_1, \dots, C_n]\!]^{\mathrm{c}}$ is deadlock free by locally considering the relationship of each C_i with the whole cycle. The idea is that C_i interoperates with $C_1, \dots, C_{i-1}, C_{i+1}, \dots, C_n$ if the parallel composition of the closed interacting semantics of the n AEIs projected on the interactions with C_i only is weakly bisimulation equivalent to the closed interacting

[6] They also demonstrate that a deadlock related architectural mismatch can be due not only to a lack of coordination among software components, but also to the presence of previously undetected errors in the software components or to a wrong choice of the software components to build up a system.

semantics of C_i. Intuitively, this means that inserting C_i into the cycle does not alter the behavior of C_i, i.e. that the behavior of the cycle assumed by C_i matches the actual behavior of the cycle. This interoperability condition is similar in spirit to the compatibility condition. Instead of comparing the overall behavior of two AEIs in a star topology with the behavior of one of them, we now compare the overall behavior of a cycle of AEIs with the behavior of one of them. We are obliged to include all the AEIs in the cycle, and not only those attached to the AEI C_i under consideration, because, as we have seen with the guest analogy, the interactions among the AEIs in the cycle not attached to C_i can give rise to mismatches between C_i and the AEIs in the cycle attached to it.

Definition 5.1. Given an architectural type, let $C_1, \ldots, C_n$ be AEIs forming a cycle. C_i is said to interoperate with $C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n$ iff

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

Theorem 5.2. *Given an architectural type, let $C_1, \ldots, C_n$ be AEIs forming a cycle. If there exists C_i such that $[\![C_i]\!]^{\mathrm{c}}_{C_1,\ldots,C_n}$ is deadlock free and C_i interoperates with $C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n$, then $[\![C_1, \ldots, C_n]\!]^{\mathrm{c}}$ is deadlock free.*

Proof. A straightforward consequence of the fact that $\approx_{\mathrm{B}}$ preserves deadlock freedom. □

If we let $\mathcal{S}(G, H, W) = \{a\_orange, a\_pineapple, t\_orange, t\_pineapple, s\_orange, s\_pineapple\}$, then for the guest analogy we have

$$\begin{aligned}
[\![G, H, W]\!]^{\mathrm{c}} / \{t\_orange, t\_pineapple\} \not\approx_{\mathrm{B}} \; & Guest[ask\_orange \mapsto a\_orange, \\
& \quad ask\_pineapple \mapsto a\_pineapple, \\
& \quad served\_orange \mapsto s\_orange, \\
& \quad served\_pineapple \mapsto s\_pineapple] \\
[\![G, H, W]\!]^{\mathrm{c}} / \{s\_orange, s\_pineapple\} \not\approx_{\mathrm{B}} \; & Host[asked\_orange \mapsto a\_orange, \\
& \quad asked\_pineapple \mapsto a\_pineapple, \\
& \quad tell\_orange \mapsto t\_orange, \\
& \quad tell\_pineapple \mapsto t\_pineapple] \\
[\![G, H, W]\!]^{\mathrm{c}} / \{a\_orange, a\_pineapple\} \not\approx_{\mathrm{B}} \; & Waiter[told\_orange \mapsto t\_orange, \\
& \quad told\_pineapple \mapsto t\_pineapple, \\
& \quad serve\_orange \mapsto s\_orange, \\
& \quad serve\_pineapple \mapsto s\_pineapple]
\end{aligned}$$

Since this system is very simple, by direct inspection we see that the problem is in the behavior of H. If $Host$ is redefined as follows

$$\begin{aligned}
Host \overset{\Delta}{=} \; & asked\_orange.tell\_orange.Host \, + \\
& asked\_pineapple.tell\_pineapple.Host
\end{aligned}$$

then we get

$$\begin{aligned}
[\![G,H,W]\!]^{\mathrm{c}}/\{t\_orange, t\_pineapple\} &\approx_{\mathrm{B}} \; Guest[ask\_orange \mapsto a\_orange,\\
&\qquad ask\_pineapple \mapsto a\_pineapple,\\
&\qquad served\_orange \mapsto s\_orange,\\
&\qquad served\_pineapple \mapsto s\_pineapple]\\
[\![G,H,W]\!]^{\mathrm{c}}/\{s\_orange, s\_pineapple\} &\approx_{\mathrm{B}} \; Host[asked\_orange \mapsto a\_orange,\\
&\qquad asked\_pineapple \mapsto a\_pineapple,\\
&\qquad tell\_orange \mapsto t\_orange,\\
&\qquad tell\_pineapple \mapsto t\_pineapple]\\
[\![G,H,W]\!]^{\mathrm{c}}/\{a\_orange, a\_pineapple\} &\not\approx_{\mathrm{B}} \; Waiter[told\_orange \mapsto t\_orange,\\
&\qquad told\_pineapple \mapsto t\_pineapple,\\
&\qquad serve\_orange \mapsto s\_orange,\\
&\qquad serve\_pineapple \mapsto s\_pineapple]
\end{aligned}$$

hence the newly obtained architectural type is deadlock free. Note that W does not interoperate with G and H. This does not cause any problem, as far as deadlock freedom is concerned. In fact, it can be proved that $[\![G,H,W]\!]^{\mathrm{c}}/\{a\_orange, a\_pineapple\}$ and $Waiter[told\_orange \mapsto t\_orange, told\_pineapple \mapsto t\_pineapple, serve\_orange \mapsto s\_orange, serve\_pineapple \mapsto s\_pineapple]$ can execute the same sequences of observable actions, i.e. they are trace equivalent. Here the point is that the two terms above have different branching points, hence $\approx_{\mathrm{B}}$ distinguishes them.

From a methodological viewpoint, given a cyclic architectural type the architectural interoperability check must be applied to each cycle $C_1, \dots, C_n$ of AEIs as follows. First, we identify the AEIs in the cycle whose closed interacting semantics is deadlock free. Then, we apply the interoperability condition to each of them until we find one that interoperates with the other AEIs in the cycle. Similarly to the case of the architectural compatibility check, Thm. 5.2 only provides a sufficient condition for deadlock freedom, for which the probability of being satisfied strongly depends on the adopted notion of observational equivalence. Therefore, a violation of the interoperability condition for a cycle of AEIs does not necessarily imply that the cycle can deadlock. However, it is wise for the architect to interpret the presence of no interoperating deadlock free AEIs along the cycle as a warning of the possible presence of some kind of mismatch in the cycle, and to consequently conduct a deeper analysis of the interactions among the AEIs in the cycle. Such a further analysis can be accomplished by repeatedly shrinking the cycle until the source of some architectural mismatch is localized. In a generic shrinking step, we consider an AEI C_i in the cycle that does not interoperate with the other AEIs in the cycle. The cause of a possible mismatch 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 temporal logic based diagnostic information coming from the failure of the verification of the interoperability condition. If we discover that an architectural mismatch exists and that its source is within C_i, then we repair C_i and we repeat the architectural interoperability check, otherwise we shrink the cycle as follows. Supposed that in the cycle C_i is preceded by C_{i-1} and followed by C_{i+1}, we replace 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, and whose interactions are the relabeled local interactions of the three original AEIs excluded those originally used to attach the three AEIs to each other. The new AEI is then checked for deadlock freedom and interoperability.

Given a cyclic topology composed of $C_1, \dots, C_n$, applying the architectural interoperability check is more convenient than directly checking $[\![C_1, \dots, C_n]\!]^{\mathrm{c}}$ for deadlock freedom. The first reason is that the application of the compatibility check requires the construction of n state spaces of the form $[\![C_1, \dots, C_n]\!]^{\mathrm{c}} / (Act - \{\tau\} - \mathcal{S}(C_i; C_1, \dots, C_n))$ where only the interactions of C_i attached to the interactions of the other AEIs in the cycle are observable. If a state space of this form is built compositionally and minimized at each step using $\approx_{\mathrm{B}}$, then its construction can be much faster than that of the state space of $[\![C_1, \dots, C_n]\!]^{\mathrm{c}}$, as this term has many more observable actions. The second reason is that, in case of deadlock, the direct check provides the architect with no diagnostic information about the deadlock source. On the contrary, the architectural interoperability check can help pinpointing potential sources of deadlock through the associated cycle shrinking procedure. The effectiveness of the cycle shrinking procedure depends on the order in which the AEIs in the cycle are considered. The sooner it is discovered an AEI in the cycle that causes deadlock, the more effective is the cycle shrinking procedure.

5.2 Related Work

To the best of our knowledge, there is no similar formal technique for detecting architectural mismatches in case of software components forming a cyclic topology. As an example, this kind of architectural mismatch is taken into account not even by the algorithm of [Inverardi et al. 2000; Inverardi and Uchitel 2001], as it does not consider the interactions within the sequence of actual behaviors matching an assumed behavior.

5.3 Example: A Cruise Control System

In this section we assess the validity of our architectural interoperability check to discover a known architectural mismatch in the cruise control system examined in [Kramer and Magee 1997].

An automobile cruise control system is controlled by three buttons: on, off, and resume. When the engine is turned on, the previous speed setting is cleared. When the engine is running and on is pressed, the cruise control system records the current speed and maintains the automobile at that speed. When the accelerator, brake, or off is pressed, the cruise control system disengages but retains the speed setting, so that if resume is pressed later on, then the system is able to accelerate or deaccelerate the automobile back to the previously recorded speed.

The flow graph of the corresponding PADL description is depicted in Fig. 4; its reduced version contains a cycle composed of S, CC, SD, SC. The behavior of each AET is shown in Table V. The underlying state transition graph has 84 states and 230 transitions (143 observable, 87 invisible).

If we let $\mathcal{S}(S, CC, SD, SC) = \{t\_engine\_on, p\_accelerator, p\_brake, p\_on, p\_off, p\_resume, t\_engine\_off, t\_clear\_speed, t\_enable\_speed\_control, t\_record\_speed, t\_disable\_speed\_control, s\_speed\}$, when applying the architectural interoperability check to the cycle composed of S, CC, SD, SC we discover that

$$\begin{array}{lll}
\llbracket S, CC, SD, SC \rrbracket^{\mathrm{c}} & \not\approx_{\mathrm{B}} & \mathit{Sensor} \\
\quad /\{\mathit{t\_clear\_speed}, & & [\mathit{turn\_engine\_on} \mapsto \mathit{t\_engine\_on}, \\
\qquad \mathit{t\_enable\_speed\_control}, & & \ \mathit{press\_accelerator} \mapsto \mathit{p\_accelerator}, \\
\qquad \mathit{t\_record\_speed}, & & \ \mathit{press\_brake} \mapsto \mathit{p\_brake}, \\
\qquad \mathit{t\_disable\_speed\_control}, & & \ \mathit{press\_on} \mapsto \mathit{p\_on}, \\
\qquad \mathit{s\_speed}\} & & \ \mathit{press\_off} \mapsto \mathit{p\_off}, \\
 & & \ \mathit{press\_resume} \mapsto \mathit{p\_resume}, \\
 & & \ \mathit{turn\_engine\_off} \mapsto \mathit{t\_engine\_off}]
\end{array}$$

$$\begin{array}{lll}
\llbracket S, CC, SD, SC \rrbracket^{\mathrm{c}} & \not\approx_{\mathrm{B}} & \mathit{CruiseController} \\
\quad /\{\mathit{s\_speed}\} & & [\mathit{turn\_engine\_on} \mapsto \mathit{t\_engine\_on}, \\
 & & \ \mathit{press\_accelerator} \mapsto \mathit{p\_accelerator}, \\
 & & \ \mathit{press\_brake} \mapsto \mathit{p\_brake}, \\
 & & \ \mathit{press\_on} \mapsto \mathit{p\_on}, \\
 & & \ \mathit{press\_off} \mapsto \mathit{p\_off}, \\
 & & \ \mathit{press\_resume} \mapsto \mathit{p\_resume}, \\
 & & \ \mathit{turn\_engine\_off} \mapsto \mathit{t\_engine\_off}, \\
 & & \ \mathit{trigger\_clear\_speed} \mapsto \mathit{t\_clear\_speed}, \\
 & & \ \mathit{trigger\_enable\_speed\_control} \mapsto \mathit{t\_enable\_speed\_control}, \\
 & & \ \mathit{trigger\_record\_speed} \mapsto \mathit{t\_record\_speed}, \\
 & & \ \mathit{trigger\_disable\_speed\_control} \mapsto \mathit{t\_disable\_speed\_control}]
\end{array}$$

$$\begin{array}{lll}
\llbracket S, CC, SD, SC \rrbracket^{\mathrm{c}} & \not\approx_{\mathrm{B}} & \mathit{SpeedDetector} \\
\quad /\{\mathit{p\_accelerator}, & & \qquad /\{\mathit{measure\_speed}\} \\
\qquad \mathit{p\_brake}, & & \qquad [\mathit{turned\_engine\_on} \mapsto \mathit{t\_engine\_on}, \\
\qquad \mathit{p\_on}, & & \qquad \ \mathit{turned\_engine\_off} \mapsto \mathit{t\_engine\_off}, \\
\qquad \mathit{p\_off}, & & \qquad \ \mathit{signal\_speed} \mapsto \mathit{s\_speed}] \\
\qquad \mathit{p\_resume}, & & \\
\qquad \mathit{t\_clear\_speed}, & & \\
\qquad \mathit{t\_enable\_speed\_control}, & & \\
\qquad \mathit{t\_record\_speed}, & & \\
\qquad \mathit{t\_disable\_speed\_control}\} & &
\end{array}$$

$$\begin{array}{lll}
\llbracket S, CC, SD, SC \rrbracket^{\mathrm{c}} & \not\approx_{\mathrm{B}} & \mathit{SpeedController} \\
\quad /\{\mathit{t\_engine\_on}, & & /\{\mathit{enable\_speed\_control}, \\
\qquad \mathit{p\_accelerator}, & & \ \ \mathit{record\_speed}, \\
\qquad \mathit{p\_brake}, & & \ \ \mathit{maintain\_speed}, \\
\qquad \mathit{p\_on}, & & \ \ \mathit{clear\_speed}, \\
\qquad \mathit{p\_off}, & & \ \ \mathit{disable\_speed\_control}, \\
\qquad \mathit{p\_resume}, & & \ \ \mathit{adjust\_throttle}\} \\
\qquad \mathit{t\_engine\_off}\} & & [\mathit{triggered\_clear\_speed} \mapsto \mathit{t\_clear\_speed}, \\
 & & \ \mathit{triggered\_enable\_speed\_control} \mapsto \mathit{t\_enable\_speed\_control}, \\
 & & \ \mathit{triggered\_record\_speed} \mapsto \mathit{t\_record\_speed}, \\
 & & \ \mathit{triggered\_disable\_speed\_control} \mapsto \mathit{t\_disable\_speed\_control}, \\
 & & \ \mathit{signalled\_speed} \mapsto \mathit{s\_speed}]
\end{array}$$

Despite the negative outcome of the architectural interoperability check, the state transition graph of the whole cycle is deadlock free. However, because of such a

Table V. AET behaviors for the cruise control system

$$\begin{array}{rcl}
\mathit{Sensor} & \triangleq & \mathit{turn\_engine\_on}.\mathit{Sensor}' \\
\mathit{Sensor}' & \triangleq & \mathit{press\_accelerator}.\mathit{Sensor}' \; + \\
 & & \mathit{press\_brake}.\mathit{Sensor}' \; + \\
 & & \mathit{press\_on}.\mathit{Sensor}' \; + \\
 & & \mathit{press\_off}.\mathit{Sensor}' \; + \\
 & & \mathit{press\_resume}.\mathit{Sensor}' \; + \\
 & & \mathit{turn\_engine\_off}.\mathit{Sensor}
\end{array}$$

$$\begin{array}{rcl}
\mathit{CruiseController} & \triangleq & \mathit{Inactive} \\
\mathit{Inactive} & \triangleq & \mathit{turned\_engine\_on}.\mathit{trigger\_clear\_speed}.\mathit{Active} \\
\mathit{Active} & \triangleq & \mathit{pressed\_accelerator}.\mathit{Active} \; + \\
 & & \mathit{pressed\_brake}.\mathit{Active} \; + \\
 & & \mathit{pressed\_on}.\mathit{trigger\_enable\_speed\_control}.\mathit{trigger\_record\_speed}.\mathit{Cruising} \; + \\
 & & \mathit{pressed\_off}.\mathit{Active} \; + \\
 & & \mathit{pressed\_resume}.\mathit{Active} \; + \\
 & & \mathit{turned\_engine\_off}.\mathit{Inactive} \\
\mathit{Cruising} & \triangleq & \mathit{pressed\_accelerator}.\mathit{trigger\_disable\_speed\_control}.\mathit{StandBy} \; + \\
 & & \mathit{pressed\_brake}.\mathit{trigger\_disable\_speed\_control}.\mathit{StandBy} \; + \\
 & & \mathit{pressed\_on}.\mathit{trigger\_enable\_speed\_control}.\mathit{trigger\_record\_speed}.\mathit{Cruising} \; + \\
 & & \mathit{pressed\_off}.\mathit{trigger\_disable\_speed\_control}.\mathit{StandBy} \; + \\
 & & \mathit{pressed\_resume}.\mathit{Cruising} \; + \\
 & & \mathit{turned\_engine\_off}.\mathit{Inactive} \\
\mathit{StandBy} & \triangleq & \mathit{pressed\_accelerator}.\mathit{trigger\_disable\_speed\_control}.\mathit{StandBy} \; + \\
 & & \mathit{pressed\_brake}.\mathit{trigger\_disable\_speed\_control}.\mathit{StandBy} \; + \\
 & & \mathit{pressed\_on}.\mathit{trigger\_enable\_speed\_control}.\mathit{trigger\_record\_speed}.\mathit{Cruising} \; + \\
 & & \mathit{pressed\_off}.\mathit{trigger\_disable\_speed\_control}.\mathit{StandBy} \; + \\
 & & \mathit{pressed\_resume}.\mathit{trigger\_enable\_speed\_control}.\mathit{Cruising} \; + \\
 & & \mathit{turned\_engine\_off}.\mathit{Inactive}
\end{array}$$

$$\begin{array}{rcl}
\mathit{SpeedDetector} & \triangleq & \mathit{turned\_engine\_on}.\mathit{WheelRevCounter} \\
\mathit{WheelRevCounter} & \triangleq & \mathit{measure\_speed}.\mathit{signal\_speed}.\mathit{WheelRevCounter} \; + \\
 & & \mathit{turned\_engine\_off}.\mathit{SpeedDetector}
\end{array}$$

$$\begin{array}{rcl}
\mathit{SpeedController} & \triangleq & \mathit{Disabled} \\
\mathit{Disabled} & \triangleq & \mathit{signalled\_speed}.\mathit{Disabled} \; + \\
 & & \mathit{triggered\_clear\_speed}.\mathit{clear\_speed}.\mathit{Disabled} \; + \\
 & & \mathit{triggered\_enable\_speed\_control}.\mathit{enable\_speed\_control}.\mathit{Enabled} \; + \\
 & & \mathit{triggered\_disable\_speed\_control}.\mathit{disable\_speed\_control}.\mathit{Disabled} \\
\mathit{Enabled} & \triangleq & \mathit{signalled\_speed}.\mathit{maintain\_speed}.\mathit{adjust\_throttle}.\mathit{Enabled} \; + \\
 & & \mathit{triggered\_enable\_speed\_control}.\mathit{enable\_speed\_control}.\mathit{Enabled} \; + \\
 & & \mathit{triggered\_record\_speed}.\mathit{record\_speed}.\mathit{Enabled} \; + \\
 & & \mathit{triggered\_disable\_speed\_control}.\mathit{disable\_speed\_control}.\mathit{Disabled}
\end{array}$$

$$\begin{array}{rcl}
\mathit{Throttle} & \triangleq & \mathit{adjusted\_throttle}.\mathit{Throttle}
\end{array}$$

negative outcome, we can suspect that some mismatch exists, which is actually the case. In fact, the reason why the cycle is deadlock free is that speed signalling related activities, which are formalized by the attachment from $SD.\mathit{signal\_speed}$ to $SC.\mathit{signalled\_speed}$, take place endlessly as long as the engine is running. But if we hide those activities, a deadlock shows up.

In order to localize the source of deadlock, let us apply the shrinking procedure to the cycle composed of S, CC, SD, SC. If we start from S, we get the following tem-

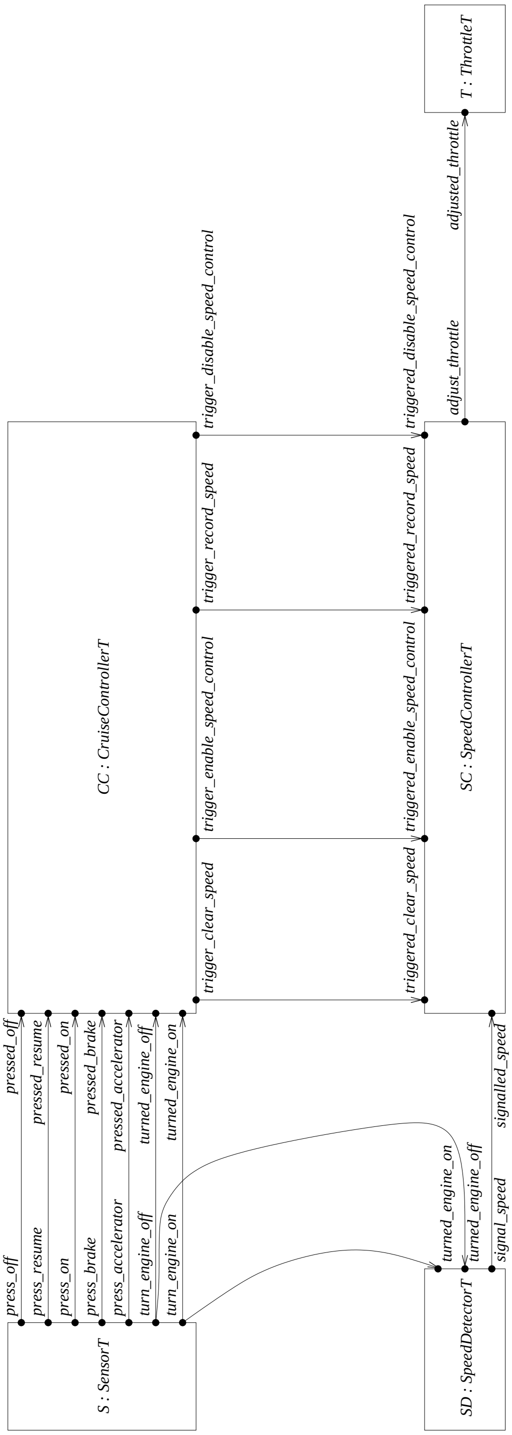

Fig. 4. Flow graph of the cruise control system

Table VI. Correct cruising behavior for CC

$$\begin{array}{lll} \mathit{Cruising} & \stackrel{\Delta}{=} & \mathit{pressed\_accelerator.trigger\_disable\_speed\_control.StandBy} + \\ & & \mathit{pressed\_brake.trigger\_disable\_speed\_control.StandBy} + \\ & & \mathit{pressed\_on.trigger\_enable\_speed\_control.trigger\_record\_speed.Cruising} + \\ & & \mathit{pressed\_off.trigger\_disable\_speed\_control.StandBy} + \\ & & \mathit{pressed\_resume.Cruising} + \\ & & \mathit{turned\_engine\_off.trigger\_disable\_speed\_control.Inactive} \end{array}$$

poral logic based diagnostic information from the failure of the weak bisimulation equivalence checking when verifying the interoperability of S w.r.t. CC, SD, SC: formula

$$[[t\_engine\_on]]\langle\langle p\_on\rangle\rangle\langle\langle t\_engine\_off\rangle\rangle\langle\langle t\_engine\_on\rangle\rangle[[p\_accelerator]]\mathit{ff}$$

is satisfied by $[\![S, CC, SD, SC]\!]^{c}$ with all the interactions not involving S being hidden, while it is not satisfied by $[\![S]\!]^{c}_{S,CC,SD,SC}$. The formula above means that, whenever t_engine_on is performed, it is possible to perform p_on, t_engine_off, and t_engine_on, thereby reaching a state in which it is not possible to perform $p\_accelerator$. This formula represents an error because, after turning the engine on, it should always be possible to press the accelerator. Since the overall cycle satisfies this formula while S does not, we conclude that the error is in one of CC, SD, SC. If we look at the behavior of CC to understand how it contributes to the satisfaction of the formula, we realize that CC is not able to perform t_clear_speed in cooperation with SC after the engine is turned on again, because CC did not disable the speed control when the engine was turned off in cruising mode. If *Cruising* is modified as shown in Table VI, then S, CC, and SD satisfy the interoperability condition and we can conclude by virtue of Thm. 5.2 that the cycle composed of S, CC, SD, SC is deadlock free. We observe that SC does not satisfy the interoperability condition, because its behavior simply models the two logical states of the speed control mechanism and the signals that can be received in each of the two states, without considering the order in which such signals can be received. Therefore, this lack of interoperability does not cause any mismatch. From the positive outcome of the interoperability check and the compatibility of T with SC, it follows that the whole architectural type is deadlock free. We conclude by observing that we have been able to systematically discover the subtle mismatch above thanks to our architectural interoperability check, whereas in [Kramer and Magee 1997] it is mentioned that the same error went originally undetected and was discovered for the first time while animating the specification.

6. ARCHITECTURAL TYPES AND ARCHITECTURAL CONFORMITY CHECK

In this section we investigate whether the architectural compatibility and interoperability checks scale from single software architectures to architectural styles. This is a desirable property, as it makes it possible to study the presence of architectural mismatches for the whole family of architectures constituting an architectural style on a single instance of the family. Investigating the scalability of the architectural compatibility and interoperability checks requires the formalization of the concept of architectural style and the provision of a solution to the membership problem for an architectural style. Unfortunately, this task is complicated by the presence of two degrees of freedom within the family of instances of the style: variability of the internal behavior of the components and variability of the topology formed by the components. As a first step towards the formalization of architectural styles, in this section we propose an intermediate abstraction called architectural type, whose instances differ only for the internal behavior of their components, while sharing the same topology. This restriction allows us to define an efficient check to verify whether a software architecture is an instance of an architectural type.

6.1 Conformity Condition

The definition of an architectural type is given by a PADL description. If we view the AETs and the architectural interactions of an architectural type as being its formal parameters, similarly to the programming languages all the instances of an architectural type can be obtained through an invocation mechanism. When invoking an architectural type, one has to specify the name of the invoked architectural type and to pass the actual, observable behavior preserving AETs and the actual names for the architectural interactions. [7]

In order to make sure that an architectural type invocation conforms to an architectural type definition, we must check whether their semantics are weakly bisimulation equivalent up to a suitable injective relabeling of their interactions. [8] In principle, this requires the construction of the state space of the architectural type definition and the architectural type invocation, whose cost grows exponentially with the number of AEIs. We show below that there exists a more efficient way to verify the conformity of an architectural type invocation to an architectural type definition, whose cost grows linearly with the number of AETs. Such a technique consists of verifying whether each actual AET is weakly bisimulation equivalent to the corresponding formal AET up to an injective relabeling of their interactions, and exploits the fact that $\approx_{\rm B}$ is a congruence w.r.t. the parallel composition operator.

Definition 6.1. Let $\mathcal{A}(\mathcal{C}'_1, \ldots, \mathcal{C}'_m; a'_1, \ldots, a'_l)$ be an invocation of the architectural type $\mathcal{A}$ defined with formal AETs $\mathcal{C}_1, \ldots, \mathcal{C}_m$ and formal architectural interactions $a_1, \ldots, a_l$. $\mathcal{C}'_i$ is said to conform to $\mathcal{C}_i$ iff there exist an injective relabeling function φ'_i for the interactions of $\mathcal{C}'_i$ and an injective relabeling function φ_i for the interactions of $\mathcal{C}_i$ such that $[\![\mathcal{C}'_i]\!][\varphi'_i] \approx_{\rm B} [\![\mathcal{C}_i]\!][\varphi_i]$. ■

If the actual AETs conform to the formal AETs, the semantics of the architectural type invocation is simply defined to be the semantics of the architectural type definition, where each formal architectural interaction is relabeled to the corresponding actual architectural interaction. Note that this definition fully abstracts from the actual AEIs of the architectural type invocation.

Definition 6.2. Let $\mathcal{A}(\mathcal{C}'_1, \ldots, \mathcal{C}'_m; a'_1, \ldots, a'_l)$ be an invocation of the architectural type $\mathcal{A}$ defined with formal AETs $\mathcal{C}_1, \ldots, \mathcal{C}_m$ and formal architectural interactions $a_1, \ldots, a_l$. If $\mathcal{C}'_i$ conforms to $\mathcal{C}_i$ for all $i = 1, \ldots, m$, then the semantics of the architectural type invocation is defined by

[7] If absent, the actual parameters are assumed to coincide with the formal ones.

[8] The relabeling must be injective in order to prevent different interactions from collapsing into a single one, which would make the check meaningless.

$$[\![\mathcal{A}(\mathcal{C}'_1,\dots,\mathcal{C}'_m;a'_1,\dots,a'_l)]\!] = [\![\mathcal{A}]\!][a_1 \mapsto a'_1,\dots,a_l \mapsto a'_l] \qquad \blacksquare$$

The next theorem shows that the definition above is correct, in the sense that under the conformity assumption we are allowed to abstract from the actual AEIs of the architectural type invocation.

Theorem 6.3. *Let $\mathcal{A}(\mathcal{C}'_1,\dots,\mathcal{C}'_m;a'_1,\dots,a'_l)$ be an invocation of the architectural type $\mathcal{A}$ defined with formal AETs $\mathcal{C}_1,\dots,\mathcal{C}_m$ and formal architectural interactions $a_1,\dots,a_l$. Let $C'_1,\dots,C'_n$ be the AEIs of the architectural type invocation. If $\mathcal{C}'_i$ conforms to $\mathcal{C}_i$ for all $i = 1,\dots,m$, then there exist an injective relabeling function φ' for the interactions of the architectural type invocation and a relabeling function φ for the interactions of the architectural type definition, with φ being injective at least on the local interactions, such that $[\![C'_1,\dots,C'_n]\!]^{\circ}[\varphi'] \approx_{\mathrm{B}} [\![\mathcal{A}]\!][\varphi]$.*

Proof. From the conformity hypothesis on the AETs, we have that each actual AEI conforms to a formal AEI. The result then follows from the congruence property of $\approx_{\mathrm{B}}$ w.r.t. the parallel composition operator. □

Note that in the theorem above we do not require the injectivity of φ on the architectural interactions of the architectural type definition. The reason is that the architectural type invocation may give the same actual name to several formal architectural interactions, which must therefore be relabeled in the same way in order for the result to hold. A straightforward consequence of the theorem is the scalability of the architectural compatibility and interoperability checks, i.e. the fact that all the instances of an architectural type possess the same compatibility and interoperability properties.

Corollary 6.4. *Let $\mathcal{A}(\mathcal{C}'_1,\dots,\mathcal{C}'_m;a'_1,\dots,a'_l)$ be an invocation of the architectural type $\mathcal{A}$ defined with formal AETs $\mathcal{C}_1,\dots,\mathcal{C}_m$ and formal architectural interactions $a_1,\dots,a_l$. If $\mathcal{C}'_i$ conforms to $\mathcal{C}_i$ for all $i = 1,\dots,m$, then the architectural type invocation and the architectural type definition have the same compatibility and interoperability properties.*

We conclude with an example. Consider the following invocation of the architectural type *PipeFilter* defined in Sect. 3, where a perfect filter type that never fails is adopted and the two architectural output interactions are given the same actual name:

PipeFilter(*PerfectFilterT*, ; *accept_request*, *generate_outcome*, *generate_outcome*)
with

```
elem_type      PerfectFilterT
  behavior     PerfectFilter ≜ accept_item.PerfectFilter'
               PerfectFilter' ≜ accept_item.PerfectFilter'' +
                                serve_item.PerfectFilter
               PerfectFilter'' ≜ serve_item.PerfectFilter'
  interactions input accept_item
               output serve_item
```

Denoted with φ_{Act} the identical relabeling function over *Act*, we have that

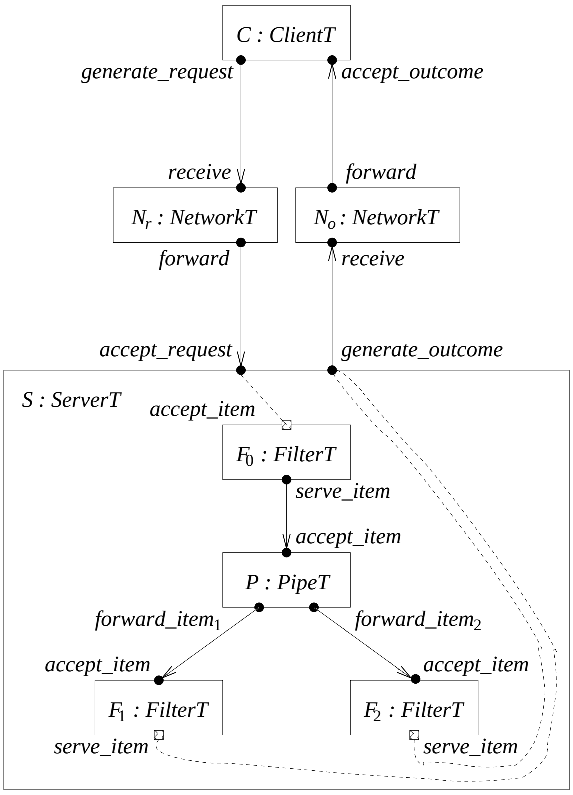

Fig. 5. Flow graph of *ClientServer*

$$\mathit{PerfectFilter}[\varphi_{Act}] \approx_{\rm B} \mathit{Filter} / \{\mathit{fail}, \mathit{repair}\}[\varphi_{Act}]$$

By virtue of Thm. 6.3, we can infer that the invocation of *PipeFilter* conforms to its definition. By virtue of Cor. 6.4, we can derive that the invocation of *PipeFilter* is deadlock free.

6.2 Hierarchical Modeling

The architectural type invocation mechanism, together with the possibility of defining architectural interactions, can be exploited to model software architectures in a hierarchical way. This is simply achieved by extending the syntax for PADL in such a way that the behavior of an AET can be defined not only through a family of sequential PA terms, but also through an invocation of a previously defined architectural type. If the behavior of an AET is defined through an architectural type invocation, the interactions of that AET are given by the actual names for the architectural interactions specified in the invocation. The semantics of that AET and its instances is then given by the semantics of the architectural type invocation, with all the local interactions of the invoked architectural type being hidden as they are internal activities from the AET viewpoint. As far as the architectural compatibility and interoperability checks are concerned, they must be applied in a

Table VII. Textual description of *ClientServer*

archi_type	$ClientServer$
archi_elem_types	
elem_type	$ClientT$
behavior	$Client \overset{\Delta}{=} generate\_request.accept\_outcome.Client$
interactions	**output** $generate\_request$
	input $accept\_outcome$
elem_type	$ServerT$
behavior	$Server \overset{\Delta}{=} PipeFilter(PerfectFilterT, ;$ $accept\_request,$ $generate\_outcome, generate\_outcome)$
interactions	**input** $accept\_request$
	output $generate\_outcome$
elem_type	$NetworkT$
behavior	$Network \overset{\Delta}{=} receive.forward.Network$
interactions	**input** $receive$
	output $forward$
archi_topology	
archi_elem_instances	$C : ClientT$
	$S : ServerT$
	$N_r, N_o : NetworkT$
archi_interactions	
archi_attachments	**from** $C.generate\_request$ **to** $N_r.receive$
	from $N_r.forward$ **to** $S.accept\_request$
	from $S.generate\_outcome$ **to** $N_o.receive$
	from $N_o.forward$ **to** $C.accept\_outcome$
end	

hierarchical way starting from the lowest level architectural types, i.e. the invoked architectural types that do not contain invocations themselves. Whenever an architectural type passes a check, then it is collapsed into a single AEI that is part of a higher level architectural type. The state space of such an AEI can be efficiently built out of the state spaces of the AEIs of the invoked architectural type if we take a compositional approach based on hiding the local interactions and minimizing at each step through $\approx_{\rm B}$.

As an example of hierarchical modeling, consider the pipe-filter system of Sect. 3 with a perfect filter type and suppose that it is the architecture of the server of a client-server system. The flow graph description of such a hierarchical architectural type is depicted in Fig. 5, while its textual description is reported in Table VII. The behavior of *ServerT* is defined through an invocation of the previously defined architectural type *PipeFilter*, where the actual AET *PerfectFilterT* substitutes for the formal AET *FilterT* and the actual names $accept\_request$, $generate\_outcome$, and $generate\_outcome$ substitute for the formal architectural interactions $F_0.accept\_item$, $F_1.serve\_item$, and $F_2.serve\_item$, respectively. Since *PerfectFilterT* conforms to *FilterT*, the architectural type *ClientServer* is well defined. Let us investigate the presence of architectural mismatches within *ClientServer*. From the architectural compatibility check and the architectural conformity check, we know that the invocation of the architectural type *PipeFilter* is deadlock free. Let us collapse this architectural type invocation into a single AEI, which corresponds to S of type *ServerT*. Since C is deadlock free and interoperates with N_r, S, and N_o, we can conclude that *ClientServer* is deadlock free.

6.3 Related Work

Some papers have appeared in the literature that address the formalization of the architectural styles. In [Abowd et al. 1995] a formal framework has been provided for precisely defining architectural styles and analyzing within and between different architectural styles. This is accomplished by means of a small set of mappings from the syntactic domain of architectural descriptions to the semantic domain of architectural meaning, following the standard denotational approach developed for programming languages. In [Dean and Cordy 1995] a syntactic theory of software architecture has been presented that is based on set theory, regular expressions, and context free grammars. Architectural styles have been categorized through the typing of the nodes and the connections in the diagrammatic syntax as well as a pattern matching mechanism. In [Moriconi et al. 1995] architectural styles have been represented as logical theories and a method has been introduced for the stepwise refinement of an abstract architecture into a relatively correct lower level architecture. Unlike the approaches above, our approach has a behavioral and algebraic nature and has been developed in such a way that it is consistent with the architectural compatibility and interoperability checks previously introduced.

Our approach to formalizing architectural styles via the intermediate abstraction of architectural types is complementary to an extension of Wright supporting architectural styles (see, e.g., [Allen and Garlan 1998]). In such an approach, the description of an architectural style only comprises the definition of component and connector types with a fixed internal behavior as well as topological constraints, whereas the component and connector instances and the related attachments are separately specified in the configurations of the style, so that the set of component and connector instances and the related attachments can vary from configuration to configuration.

7. CONCLUSION

In this paper we have developed a compositional, hierarchical and graphical ADL called PADL for the specification and analysis of families of component based software systems. PADL provides a uniform framework in which the three different causes of deadlock related architectural mismatches mentioned in the introduction can be dealt with by means of the standard weak bisimulation equivalence. Causes (1) and (2) have been treated by means of an architectural compatibility check that ensures the absence of deadlock within a set of software components interacting through a given deadlock free software component. This check, which scales to the whole architecture in case of acyclic topology, subsumes the compatibility check of [Allen and Garlan 1997] and equals the effectiveness of the algorithm of [Inverardi et al. 2000; Inverardi and Uchitel 2001]. Cause (3) has been instead treated through an architectural interoperability check that guarantees the absence of deadlock within a set of interacting software components forming a cyclic topology. Both checks, which rely on the properties of the weak bisimulation equivalence, have turned out to be more convenient – for efficiency and/or diagnosis related reasons – than directly verifying deadlock freedom for the considered set of software components. The usefulness of both checks has been assessed on a compressing proxy system and a cruise control system, respectively, with TwoTowers [Bernardo 2002], a software tool for the functional verification and performance evaluation of concurrent and distributed systems described with the stochastic process algebra EMPA$_{\mathrm{gr}}$ [Bravetti and Bernardo 2000]. Finally, we have addressed the issue of scaling the architectural compatibility and interoperability checks to architectural styles through an extension of PADL. We have proposed an approximation of an architectural style called architectural type, whose instances differ only for the internal behavior of their components. We have defined an efficient architectural conformity check based on the weak bisimulation equivalence to verify whether an architecture is an instance of an architectural type. We have shown that all the architectures conforming to the same architectural type possess the same compatibility and interoperability properties.

Two lessons have been learnt. First, the detection of architectural mismatches in the case of complex component based software systems is practically impossible without the support of a formal description of the architecture of the systems (see, e.g., the cruise control system). In this respect, the architectural descriptions benefit a lot from the formality, compositionality, and expressivity of the process algebras. Second, the process algebras are well suited at the architectural level of design not only because of their compositional modeling facilities, but also for their analysis related machinery, which enables the software architect to detect mismatches in a rather efficient and automated way and to get some diagnostic information useful for identifying the cause of the mismatches. In conclusion, we can say that the process algebra based ADLs can be viewed as members of a next generation of process algebras that offer an enhanced usability – through graphical and hierarchical modeling capabilities – and a more controlled way of modeling complex systems – through static checks like architectural compatibility, interoperability, and conformity.

As far as future work is concerned, there are several directions that we would like to explore:

(1) The deadlock freedom results can be so far scaled from the individual components to the whole architectural types only in the case of acyclic topologies. We would like to understand whether and when this form of scalability can be achieved in the case of cyclic topologies.

(2) There are deadlock free software systems made out of deadlock free components that do not pass the architectural compatibility or interoperability checks. The reason is that such checks are based on the weak bisimulation equivalence, which is known to be very sensitive to the branching points in the component behavior. We would like to weaken the architectural compatibility and interoperability checks using other standard observational equivalences lying between the weak bisimulation equivalence and the trace equivalence, and to investigate to which extent the deadlock freedom results are preserved.

(3) Deadlock is not the only kind of architectural mismatch. As seen in the introduction, in general the designer may start with components each possessing

a certain property and desire to get a system possessing a property somehow related to the properties of its components. As a consequence, we would like to investigate the adequacy of analysis techniques based on temporal logics rather than equivalence checking. As a first step, since our architectural checks are based on the weak bisimulation equivalence, we may consider its logical characterization given by the weak Hennessy-Milner logic [Milner 1989]. As a further step, we believe that compositional model checking [Clarke et al. 1989] and partial model checking [Andersen 1995] may be profitably used.

(4) The notion of architectural type that we have proposed as an approximation of the concept of architectural style does not permit any form of variability of the topology from instance to instance of the architectural type. We would like to find out a more liberal definition of the notion of architectural type, which allows for some form of topological variability while preserving the scalability of the architectural compatibility and interoperability checks. Some work in this direction has been recently done in [Bernardo and Franzè 2002a; 2002b].

(5) The designer is often faced with the problem of choosing among different software architectures that are functionally equivalent. This choice is clearly driven by nonfunctional factors, and mostly by performance requirements. We would like to develop an extension of our framework that takes performance aspects into account. Similarly to the case of the detection of functional mismatches, the extended framework should provide some diagnostic information at the architectural level whenever unexpected values for the performance measures are obtained. We believe that this can be achieved through a suitable mapping from an enhanced PADL like language to structured performance models, like queueing networks [Lavenberg 1983] and Markov chains represented through Kronecker operators [Davio 1981]. We have done some preliminary work in this direction in [Bernardo et al. 2002; Balsamo et al. 2002].

(6) Our framework should be put in the context of the whole software life cycle. For this reason, we would like to study whether and to which extent:
- a PADL description can be synthesized from a description of the user requirements, e.g. given in the UML notation;
- a PADL description can be used to automatically generate object oriented code that possesses the properties proved at the architectural level, as well as tests to be applied to the code that follows the described architecture;
- a PADL description can be used to predict the impact of the hardware platform on the functionality and the performance of the described software architecture.

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their useful comments and suggestions on an earlier version of this work.

REFERENCES

Abowd, G. D., Allen, R., and Garlan, D. 1995. Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Method. 4*, 319–364.

Allen, R. and Garlan, D. 1997. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Method. 6*, 213–249.

Allen, R. and Garlan, D. 1998. A case study in architectural modelling: the aegis system. In *Proc. of the 8th Int. Workshop on Software Specification and Design (IWSSD-8)*.

Andersen, H. R. 1995. Partial model checking. In *Proc. of the 10th IEEE Int. Symp. on Logic in Computer Science (LICS 1995)*. IEEE-CS Press, 398–407.

Balsamo, S., Bernardo, M., and Simeoni, M. 2002. Combining stochastic process algebras and queueing networks for software architecture analysis. In *Proc. of the 3rd Int. Workshop on Software and Performance (WOSP 2002)*. ACM Press, 190–202.

Bernardo, M. 2002. *TwoTowers 2.0 User Manual.* http://www.sti.uniurb.it/bernardo/twotowers/.

Bernardo, M., Ciancarini, P., and Donatiello, L. 2000. On the formalization of architectural types with process algebras. In *Proc. of the 8th ACM Int. Symp. on the Foundations of Software Engineering (FSE-8)*. ACM Press, 140–148.

Bernardo, M., Ciancarini, P., and Donatiello, L. 2001. Detecting architectural mismatches in process algebraic descriptions of software systems. In *Proc. of the 1st Working IEEE/IFIP Conf. on Software Architecture (WICSA 2001)*. IEEE-CS Press, 77–86.

Bernardo, M., Donatiello, L., and Ciancarini, P. 2002. Stochastic process algebra: from an algebraic formalism to an architectural description language. In *Performance Evaluation of Complex Systems: Techniques and Tools*, M. C. Calzarossa and S. Tucci, Eds. LNCS, vol. 2459. Springer, 236–260.

Bernardo, M. and Franzè, F. 2002a. Architectural types revisited: extensible and/or connections. In *Proc. of the 5th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2002)*. LNCS, vol. 2306. Springer, 113–128.

Bernardo, M. and Franzè, F. 2002b. Exogenous and endogenous extensions of architectural types. In *Proc. of the 5th Int. Conf. on Coordination Models and Languages (COORDINATION 2002)*. LNCS, vol. 2315. Springer, 40–55.

Bravetti, M. and Bernardo, M. 2000. Compositional asymmetric cooperations for process algebras with probabilities, priorities, and time. In *Proc. of the 1st Int. Workshop on Models for Time Critical Systems (MTCS 2000)*. ENTCS, vol. 39(3). Elsevier.

Clarke, E. M., Grumberg, O., and Peled, D. A. 1999. *Model Checking.* MIT Press.

Clarke, E. M., Long, D. E., and McMillan, K. L. 1989. Compositional model checking. In *Proc. of the 4th IEEE Int. Symp. on Logic in Computer Science (LICS 1989)*. IEEE-CS Press, 353–362.

Davio, M. 1981. Kronecker products and shuffle algebra. *IEEE Trans. Comput. 30*, 116–125.

Dean, T. R. and Cordy, J. R. 1995. A syntactic theory of software architecture. *IEEE Trans. Softw. Eng. 21*, 302–313.

DeRemer, F. and Kron, H. H. 1976. Programming-in-the-large versus programming-in-the-small. *IEEE Trans. Softw. Eng. 2*, 80–86.

Hoare, C. A. R. 1985. *Communicating Sequential Processes.* Prentice Hall.

Inverardi, P. and Uchitel, S. 2001. Proving deadlock freedom in component-based programming. In *Proc. of the 4th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2001)*. LNCS, vol. 2029. Springer, 60–75.

Inverardi, P. and Wolf, A. L. 1995. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Trans. Softw. Eng. 21*, 373–386.

Inverardi, P., Wolf, A. L., and Yankelevich, D. 2000. Static checking of system behaviors using derived component assumptions. *ACM Trans. Softw. Eng. Method. 9*, 239–272.

Kramer, J. and Magee, J. 1997. Exposing the skeleton in the coordination closet. In *Proc. of the 2nd Int. Conf. on Coordination Models and Languages (COORDINATION 1997)*. LNCS, vol. 1282. Springer, 18–31.

Lavenberg, S. S., Ed. 1983. *Computer Performance Modeling Handbook.* Academic Press.

Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. 1995. Specifying distributed software architectures. In *Proc. of the 5th European Software Engineering Conf. (ESEC 1995)*. LNCS, vol. 989. Springer, 137–153.

Milner, R. 1989. *Communication and Concurrency.* Prentice Hall.

Moriconi, M., Qian, X., and Riemenschneider, R. A. 1995. Correct architecture refinement. *IEEE Trans. Softw. Eng. 21*, 356–372.

Perry, D. E. and Wolf, A. L. 1992. Foundations for the study of software architecture. *ACM Softw. Eng. Notes 17*, 40–52.

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. 1995. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng. 21*, 314–335.

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

van Glabbeek, R. J. 2001. The linear time – branching time spectrum i. the semantics of concrete, sequential processes. In *Handbook of Process Algebra*, J. A. Bergstra, A. Ponse, and S. A. Smolka, Eds. Elsevier, 3–99.

Received July 2001; revised April 2002; accepted July 2002