Detecting Architectural Mismatches in Process Algebraic Descriptions of Software Systems

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

Paolo Ciancarini, Lorenzo Donatiello
Università di Bologna
Dipartimento di Scienze dell'Informazione
Mura Anteo Zamboni 7, 40127 Bologna, Italy
cianca, donat@cs.unibo.it

Abstract

Formalizing the description of software systems helps to detect the presence of architectural mismatches that can arise when assembling software components together. In this paper we identify three causes of architectural mismatches: 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 then show how to deal with all of them within a uniform, process algebraic framework. We begin with the first two causes by strengthening a previously defined architectural compatibility check based on observational equivalences, in order to achieve a deadlock freedom result for the set of components interacting via a certain connection. We subsequently concentrate on the third cause by defining a novel architectural interoperability check based on observational equivalences, which guarantees absence of deadlock within a set of interacting components forming a cyclic topology. We finally assess the adequacy of our architectural interoperability check by applying it to the description of a cruise control system.

1. Introduction

The software architecture level of design enables us to cope with the increasing size and complexity of software systems during the early stage of their development [12, 13]. To achieve this, the design focus is turned from algorithmic and data structure related issues to the overall architecture of a 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 for short) should be devised to formalize software architectures instead of using informal box-and-line diagrams, and companion tools should be implemented to support the automatic analysis of architectural properties in order to allow the designer to make principled choices. Among the formal method based ADLs appeared in the literature, we mention those relying on process algebras [2, 10, 3], Z [1], and the CHAM [7].

The formal description of the software architecture of a 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, which allows for the early detection of errors thus resulting in time and money saving. This paper is about the latter purpose.

Nowadays systems are typically made out of numerous components whose behavior is individually well known. Think e.g. of embedded systems or web centered applications. Thus, the main problem faced by a software architect is that of understanding whether the components coordinate, i.e. fit together well. If the architecture of a system is given a formal description, then mathematical techniques can be used to automatically prove the well formedness of the system, i.e. the absence of architectural mismatches. In this paper we deal with three causes of architectural mismatches: 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 interacting components forming a cyclic topology.

The first cause is the simplest example of architectural mismatch: think e.g. of a component that wants to send data to another component via a certain connector, with

the connector never willing to accept data to be forwarded. This kind of architectural mismatch has been considered in the process algebraic framework of WRIGHT [2]. Using WRIGHT the architect is required to specify the ports of each component and the roles of each connector, where a port represents the behavior of a component w.r.t. an interaction with a connector, while the roles of a connector describe the expected interactions of each component attached to the connector. Then, given a connector and the set of components attached to it, all the pairs of corresponding ports and roles are verified to be compatible in order to ensure an overall deadlock free interaction of the set of components through that connector, under the assumption that each involved component has a single interaction with the connector.

In [8] it has been argued that mismatches should not be checked at the port/role level only, because the internal (as opposed to interactional) behavior of a component may correlate different interactions of the component itself in such a way that the system blocks. As an example, in [8] the CHAM has been employed to model a compressing proxy, whose gzip component interacts properly with the rest of the system as long as its input flow interaction and its output flow interaction are considered 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 considering the two flows together w.r.t. the rest of the system. This kind of architectural mismatch has been dealt with in [8, 6] by extracting from each component description its actual behavior and its assumptions about the behavior of the rest of the system, and by defining an algorithm 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 [3] the same kind of mismatch has been treated in a process algebraic framework by means of standard projections and observational equivalences. The idea is to verify that, for each pair composed of a component and a connector attached to each other, the behavior of the component projected on all the interactions with the connector is observationally equivalent to the behavior of the connector projected on all the interactions with the component. The first contribution of this paper is to strengthen the architectural compatibility check of [3] in order to achieve a deadlock freedom result, thus subsuming the compatibility check of [2] and equalling the effectiveness of the algorithm of [8, 6]. The importance of such a contribution is that different techniques for detecting similar architectural mismatches are brought into a uniform, process algebraic framework using standard machinery.

The second contribution of this paper is to show that a third cause of architectural mismatches can be revealed through standard observational equivalences in a process algebraic framework. The point here is that mismatches should not be checked only between pairs of connected 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 [8, 6], 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 [8]. Here we have three 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 he/she wants an orange juice or a pineapple juice. The guest then expects that the host tells the waiter the desired drink and that the waiter brings that drink. Suppose that the binary 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. In this paper we define an architectural interoperability check in the framework of [3], which guarantees the absence of deadlock for sets of interacting components that form a cyclic topology.

This paper is organized as follows. In Sect. 2 and 3 we recall our formal framework based on process algebras, observational equivalences, and PADL [3]. In Sect. 4 we strengthen the architectural compatibility check of [3] so as to achieve a deadlock freedom result. In Sect. 5 we provide a novel architectural interoperability check in such a way that a deadlock freedom result is obtained. In Sect. 6 we consider the case of a cruise control system examined in [9] and we show that our architectural interoperability check is able to detect a well known mismatch. In Sect. 7 we report some concluding remarks.

2. Process Algebras

Process algebras [11, 5] are algebraic languages which support the compositional description of concurrent and distributed systems and the formal verification of their properties. The basic elements of any process algebra are its actions, which represent activities carried out by the systems being modeled, and its operators (among which a parallel composition operator), which are used to compose algebraic descriptions.

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

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

where a belongs to a set Act of actions including a distinguished action τ for unobservable activities, $L, S \subseteq$

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

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

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

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

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

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

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

Table 1. Operational semantics for PA

$Act - \{\tau\}$, 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 $A \triangleq E$.

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

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

Due to their algebraic nature, process description languages like PA naturally lend themselves to the definition of equivalences and preorders. In particular, in the concurrency theory literature several notions of equivalence can be found, which relate terms according to a certain interpretation of their behavior. Here we recall the weak bisimulation equivalence, which captures the ability of two terms to simulate each other behavior up to τ actions. Before introducing it, we generalize the transition relation $\stackrel{}{\longrightarrow}$ labeled with actions to the transition relation $\stackrel{}{\Longrightarrow}$ labeled with sequences of actions: $\stackrel{a_1 \ldots a_n}{\Longrightarrow} \equiv \stackrel{a_1}{\longrightarrow} \ldots \stackrel{a_n}{\longrightarrow}$, with ε denoting the empty string and $\stackrel{\varepsilon}{\Longrightarrow}$ 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 $\stackrel{\sigma}{\Longrightarrow} \equiv \stackrel{\tau^m}{\Longrightarrow} \stackrel{\sigma}{\Longrightarrow} \stackrel{\tau^n}{\Longrightarrow}$, with $m, n \in \mathbf{N}$, to indicate the execution of an action sequence σ possibly preceded and followed by the execution of arbitrarily many invisible actions.

A binary relation $\mathcal{B}$ over process terms is a weak bisimulation [11] iff, whenever $(E_1, E_2) \in \mathcal{B}$, then for all $a \in Act$:

- whenever $E_1 \stackrel{a}{\longrightarrow} E_1'$, then $E_2 \stackrel{\hat{a}}{\Longrightarrow} E_2'$ and $(E_1', E_2') \in \mathcal{B}$ for some E_2';
- whenever $E_2 \stackrel{a}{\longrightarrow} E_2'$, then $E_1 \stackrel{\hat{a}}{\Longrightarrow} E_1'$ and $(E_1', E_2') \in \mathcal{B}$ for some E_1'.

Note that $\hat{a} = a$ for $a \neq \tau$, $\hat{a} = \varepsilon$ for $a = \tau$. The union of all the weak bisimulations, denoted by $\approx_{\mathrm{B}}$, is called the weak bisimulation equivalence. $\approx_{\mathrm{B}}$ enjoys several algebraic properties w.r.t. the dynamic operators:

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

The last four properties show the ability of $\approx_{\mathrm{B}}$ to abstract from τ actions and make $\approx_{\mathrm{B}}$ suited to reason on projections (obtained through the hiding operator) of the behavior of system components. 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

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

This will be exploited in the proof of Thm. 4.3.

3. PADL

A description in PADL, the ADL based on PA introduced in [3], is an architectural type (AT). Each AT is defined as a function of its architectural element types (AETs), its topology, and its architectural interactions (AIs). An AET is in turn defined as a function of its behavior, specified either as a family of PA sequential terms or through an invocation of a previously specified AT, and its interactions, specified as a set of PA actions. The architectural topology consists of a fixed set of architectural element instances (AEIs) related by a fixed set of attachments among their interactions. Finally, AIs are interactions of AEIs that act as interactions for the whole AT, thus supporting hierarchical architectural modeling.

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

The first part of a PADL description defines the name of the AT and its formal parameters, i.e. AETs and AIs:

$\mathbf{archi\_type}\ PipeFilter(FilterT, PipeT; PipeFilterI)$

The specification above indicates that the AT named $PipeFilter$ relies on two AETs, $FilterT$ and $PipeT$, and on a set $PipeFilterI$ of AIs.

The second part of the description defines the AETs that have been specified to be formal parameters of the AT in the first part. Each AET is defined through a family of PA sequential terms (or an invocation of a previously specified AT) expressing its behavior and a set of PA actions expressing its interactions:

$\mathbf{elem\_type}\ FilterT(Filter; FilterI)$
$\quad \mathbf{behavior}\ Filter \triangleq accept\_item.Filter'$
$\qquad Filter' \triangleq accept\_item.Filter'' + serve\_item.Filter$
$\qquad Filter'' \triangleq serve\_item.Filter'$
$\quad \mathbf{interactions}\ FilterI = accept\_item, serve\_item$
$\mathbf{elem\_type}\ PipeT(Pipe, PipeI)$
$\quad \mathbf{behavior}\ Pipe \triangleq accept\_item.$
$\qquad (forward\_item_1.Pipe + forward\_item_2.Pipe)$
$\quad \mathbf{interactions}\ PipeI = accept\_item, forward\_item_1, forward\_item_2$

The specification above indicates that AET $FilterT$ has the

archi_type	⟨name and formal parameters⟩
archi_elem_types	⟨architectural element types: behaviors and interactions⟩
archi_topology	⟨architectural topology: architectural element instances and attachments⟩
archi_interactions	⟨architectural interactions⟩
end	

Table 2. PADL textual notation

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

The third part of the description defines the topology of the AT in terms of instances of the previously introduced AETs and related attachments among their interactions:

$$\begin{array}{ll} \textbf{archi\_topology} & F_0, F_1, F_2 : FilterT \\ & P : PipeT \\ & F_0.serve\_item \textbf{ to } P.accept\_item \\ & F_1.accept\_item \textbf{ to } P.forward\_item_1 \\ & F_2.accept\_item \textbf{ to } P.forward\_item_2 \end{array}$$

The attachments above specify that an instance P of $PipeT$ is connected to an upstream instance F_0 of $FilterT$ and two downstream instances F_1, F_2 of $FilterT$.

Finally, the fourth part defines the AIs. They are interactions of the previously defined AEIs that are used when defining the behavior of an AET of another AT by invoking the AT being specified, i.e. when plugging the AT at hand in the context of a larger system:

$$\begin{array}{lcl} \textbf{archi\_interactions } PipeFilterI & = & F_0.accept\_item, \\ & & F_1.serve\_item, \\ & & F_2.serve\_item \end{array}$$

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

$PipeFilter$ can be pictorially represented through the flow graph [11] of Fig. 1, where boxes denote AEIs, black circles/white squares denote interactions/AIs, and edges between ports denote attachments.

The semantics of a PADL specification is given by translation into PA as follows.

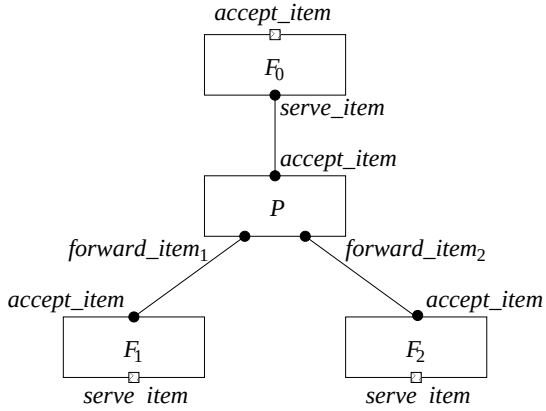

Figure 1. Flow graph of $PipeFilter$

Definition 3.1 Given a PADL specification, let C be an AET with behavior E and interactions I. The semantics of C and its instances is defined to be its behavior projected onto its interactions:

$$[\![C]\!] = E/(Act - \{\tau\} - I) \qquad \blacksquare$$

In our example we have $[\![FilterT]\!] = [\![F_0]\!] = [\![F_1]\!] = [\![F_2]\!] = Filter/(Act - \{\tau\} - FilterI)$ and $[\![PipeT]\!] = [\![P]\!] = Pipe/(Act - \{\tau\} - PipeI)$.

The semantics of an AT is then 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 example we have:

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

The use of the relabeling operator is necessary to make 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 actions to synchronize only if they are equal, in $[\![PipeFilter]\!]$ above each $serve\_item$ executed by $[\![F_0]\!]$ and

each $accept\_item$ executed by $[\![P]\!]$ is relabeled to the same action a (occurring neither in $[\![F_0]\!]$ nor in $[\![P]\!]$ to avoid interferences), then a synchronization on a is forced between the relabeled versions of $[\![F_0]\!]$ and $[\![P]\!]$ by means of operator $\|_{\{a,a_1,a_2\}}$.

To define the semantics of an arbitrary AET, we preliminarily introduce some notation that will be used also in the rest of the paper. Let $\{C_1,\ldots,C_n\}$ be a subset of its AEIs, with interactions $I_{C_1},\ldots,I_{C_n}$ respectively; we use i,j,k to range over $\{1,\ldots,n\}$. We say that $C_i.a_1$ is connected to $C_j.a_2$ iff either there is an attachment between them, or there exists an interaction a_3 of C_k such that $C_i.a_1$ is connected to $C_k.a_3$ and there is an attachment between $C_k.a_3$ and $C_j.a_2$. Moreover, we say that a subset of attachments is connected iff the involved interactions are pairwise connected via attachments of the subset only and the subset is maximal. Denoted by $I_{C_i;C_1,\ldots,C_n} \subseteq I_{C_i}$ the subset of interactions of C_i attached to $C_1,\ldots,C_n$, let $S(C_1,\ldots,C_n)$ be a set of as many fresh [1] actions as there are connected subsets of attachments among the considered AEIs of the PADL specification, let $\varphi_{C_i;C_1,\ldots,C_n} : I_{C_i;C_1,\ldots,C_n} \longrightarrow S(C_1,\ldots,C_n)$ be action relabeling functions such that $\varphi_{C_i;C_1,\ldots,C_n}(a_1) = \varphi_{C_j;C_1,\ldots,C_n}(a_2)$ iff $C_i.a_1$ is connected to $C_j.a_2$, and let $S(C_i;C_1,\ldots,C_n) = \varphi_{C_i;C_1,\ldots,C_n}(I_{C_i;C_1,\ldots,C_n})$ and $S(C_i,C_j;C_1,\ldots,C_n) = S(C_i;C_1,\ldots,C_n) \cap S(C_j;C_1,\ldots,C_n)$.

Definition 3.2 Given the PADL specification above, we define the interacting semantics $[\![C_i]\!]_{C_1,\ldots,C_n}$ of C_i restricted to $C_1,\ldots,C_n$ by

$$[\![C_i]\!]/(Act - \{\tau\} - I_{C_i;C_1,\ldots,C_n})[\varphi_{C_i;C_1,\ldots,C_n}]$$

The semantics $I(C_1,\ldots,C_n)$ of the considered group of AEIs is given by

$$[\![C_1]\!]_{C_1,\ldots,C_n} \|_{S(C_1,C_2;C_1,\ldots,C_n)} \cdots \cdots \|_{\cup_{i=1}^{n-1} S(C_i,C_n;C_1,\ldots,C_n)} [\![C_n]\!]_{C_1,\ldots,C_m} \quad \blacksquare$$

The PA term above is the semantics of a whole PADL specification whenever the considered group comprises all of its AEIs, in which case the hiding occurring in the definition of the restricted interacting semantics is redundant.

We conclude by observing that PADL is closely related to WRIGHT [2]. In particular, we note that the behavior of AETs is specified through process terms, and that the semantics is given by translation into a process term by essentially exploiting the parallel composition operator. 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 which can be interpreted as being a component or a connector depending on the particular system. This avoids redundancy in the specifications that results in 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 the interactions are simply expressed as actions (similarly to DARWIN [10]). This allows ports and roles to be retrieved via projections (see the definition of restricted interacting semantics) and avoids redundancy in the specifications. Third, PADL allows the hierarchical modeling of families of architectures and provides an efficient technique to check an architecture instance for conformity to an architectural type [3]. Fourth, PADL is equipped with a graphical notation that is standard in the concurrency theory literature.

[1] I.e., not occurring in the semantics of the considered AEIs.

4. Compatibility Check

In order to make sure that every pair of AEIs attached to each other interact in a proper way, in [3] PADL has been equipped with an architectural compatibility check. It consists of verifying that, for each pair of AEIs attached to each other, their restricted interacting semantics are weakly bisimulation equivalent. According to the notation of Def. 3.2, we say that C_i is compatible with C_j iff $[\![C_i]\!]_{C_i,C_j} \approx_{\mathrm{B}} [\![C_j]\!]_{C_i,C_j}$. In our example this condition is met by all the pairs of interacting AEIs:

$$[\![F_0]\!]/(Act - \{serve\_item,\tau\})[serve\_item \mapsto a] \approx_{\mathrm{B}} [\![P]\!]/(Act - \{accept\_item,\tau\})[accept\_item \mapsto a]$$
$$[\![F_1]\!]/(Act - \{accept\_item,\tau\})[accept\_item \mapsto a_1] \approx_{\mathrm{B}} [\![P]\!]/(Act - \{forward\_item_1,\tau\})[forward\_item_1 \mapsto a_1]$$
$$[\![F_2]\!]/(Act - \{accept\_item,\tau\})[accept\_item \mapsto a_2] \approx_{\mathrm{B}} [\![P]\!]/(Act - \{forward\_item_2,\tau\})[forward\_item_2 \mapsto a_2]$$

In general such a compatibility check does not guarantee that the interaction of the AEIs attached to a given AEI is deadlock free. In this section we consider only ATs whose reduced flow graph (in which all the attachments between two AEIs are collapsed into a single one) is acyclic; cycles will be dealt with in the next section. The absence of cycles implies that all the AEIs interacting through a given AEI cannot interact in other ways.

Definition 4.1 Given an acyclic PADL specification, let $C_1,\ldots,C_n$ be the AEIs attached to AEI K. The interaction $I(K;C_1,\ldots,C_n)$ of $C_1,\ldots,C_n$ through K is defined by

$$[\![K]\!]_{K,C_1,\ldots,C_n} \|_{S(K;K,C_1)} [\![C_1]\!]_{K,C_1,\ldots,C_n} \|_{S(K;K,C_2)} \cdots \cdots \|_{S(K;K,C_n)} [\![C_n]\!]_{K,C_1,\ldots,C_n}$$

$I(K;C_1,\ldots,C_n)$ is deadlock free iff for each of its states s there exist a visible action $a \in Act - \{\tau\}$ and a state s' such that $s \stackrel{a}{\Longrightarrow} s'$. $\blacksquare$

Deadlock freedom holds for the interactions of systems like $PipeFilter$, but if we take e.g. $[\![C]\!] \triangleq \tau.C_1 + \tau.C_2, C_1 \triangleq$

$a.C_1, C_2 \triangleq b.C_2$ and $[\![K]\!] \triangleq \tau.K_1 + \tau.K_2, K_1 \triangleq a.K_1, K_2 \triangleq b.K_2$ with equal actions attached to each other and identically relabeled, then $[\![C]\!]_{K,C} \approx_{\mathrm{B}} [\![K]\!]_{K,C}$ but $I(K;C) \equiv [\![K]\!]_{K,C} \|_{\{a,b\}} [\![C]\!]_{K,C}$ is not deadlock free even if $[\![C]\!]_{K,C}$ and $[\![K]\!]_{K,C}$ are such.

A sufficient condition for deadlock freedom can be retrieved by slightly modifying the architectural compatibility check of [3] as follows. Let us define the interacting semantics of AEI C_i w.r.t. AEI C_j by $[\![C_i]\!]'_{C_i,C_j} = [\![C_i]\!][\varphi_{C_i;C_i,C_j}]$.

Definition 4.2 Given an acyclic PADL specification, let C_i and C_j be two AEIs. C_j is said to be compatible with C_i iff

$$[\![C_i]\!]'_{C_i,C_j} \|_{S(C_i;C_i,C_j)} [\![C_j]\!]_{C_i,C_j} \approx_{\mathrm{B}} [\![C_i]\!]'_{C_i,C_j}$$

■

Theorem 4.3 Given an acyclic PADL specification, let $C_1, \ldots, C_n$ be the AEIs attached to AEI K. If $[\![K]\!]$ is deadlock free and C_i is compatible with K for all $i = 1, \ldots, n$, then $I(K; C_1, \ldots, C_n)$ is deadlock free.

Proof The result follows by proving that $I(K; C_1, \ldots, C_n) \approx_{\mathrm{B}} [\![K]\!]_{K,C_1,\ldots,C_n}$ as $[\![K]\!]_{K,C_1,\ldots,C_n}$ is deadlock free, because it differs from $[\![K]\!]$ only for the names of its interactions, and $\approx_{\mathrm{B}}$ preserves deadlock freedom. This can be proved by induction on n by observing that $[\![K]\!]'_{K,C_i} \|_{S(K;K,C_i)} [\![C_i]\!]_{K,C_i} \approx_{\mathrm{B}} [\![K]\!]'_{K,C_i} \Longrightarrow [\![K]\!]_{K,C_1,\ldots,C_n} \|_{S(K;K,C_i)} [\![C_i]\!]_{K,C_i} \approx_{\mathrm{B}} [\![K]\!]_{K,C_1,\ldots,C_n}$ for all $i = 1, \ldots, n$, because $\approx_{\mathrm{B}}$ is a congruence w.r.t. relabeling, and by exploiting the fact that $\approx_{\mathrm{B}}$ is a congruence w.r.t. parallel composition. ■

It is worth noting that verifying architectural compatibility over each pair C_i, K is more convenient than globally verifying that $I(K; C_1, \ldots, C_n)$ is deadlock free. The first reason is that the local check requires the construction of n state spaces resulting from the parallel composition of two terms only (see Def. 4.2), whereas the global check requires the construction of one state space resulting from the parallel composition of n terms (see Def. 4.1). Thus, the complexity of the local check is linear in the number of AEIs, while the complexity of the global check is exponential in the number of AEIs. The second reason is that, in case of mismatch, the local check provides the architect with diagnostic information to single out the AEIs responsible for the mismatch, while the global check does not.

We conclude by observing that this new architectural compatibility check not only strengthens that of [3], but also subsumes the compatibility check of [2] and equals the effectiveness of the algorithm of [8, 6], thus bringing into a uniform, process algebraic framework different techniques for detecting similar architectural mismatches using standard machinery.

5. Interoperability Check

Checking architectural compatibility is unfortunately not enough to guarantee that a system is deadlock free, i.e. there may be further causes of architectural mismatches. Let us consider e.g. the flow graph of Fig. 2, which represents the guest analogy mentioned in Sect. 1. Although the AEIs are deadlock free, the whole system is not. Here the problem is that *Guest* and *Waiter* do not communicate directly, so e.g. if *Guest* asks for an orange juice and *Host* tells *Waiter* to bring a pineapple juice instead, *Guest* and *Waiter* do not synchronize, thus causing the system to block. If we construct the state space, we discover that there are two deadlock states arising from the situation just mentioned and its symmetrical.

If we look carefully at the flow graph of Fig. 2, we observe that the three AEIs form a loop. The architectural mismatch arises because *Host* can act as a malicious agent between *Guest* and *Waiter*. Should the behavior of *Host* and *Waiter* be concentrated in a single AEI interacting with *Guest*, the mismatch would be detected through the architectural compatibility check.

In order to catch this kind of mismatches, in this section we define a suitable architectural interoperability check.

Definition 5.1 Given a PADL specification, let $C_1, \ldots, C_n$ be AEIs forming a loop. $C_1, \ldots, C_n$ are said to interoperate iff for all $i = 1, \ldots, n$

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

■

Theorem 5.2 Given a PADL specification, let $C_1, \ldots, C_n$ be AEIs forming a loop. If there exists $i \in \{1, \ldots, n\}$ such that $[\![C_i]\!]_{K_1,\ldots,K_m}$ is deadlock free and $C_1, \ldots, C_n$ interoperate, then $I(C_1, \ldots, C_n)$ is deadlock free.

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

Unlike the architectural compatibility check, the architectural interoperability check guarantees the absence of deadlock with a cost that is exponential in the number of AEIs in the loop. One cannot even think of reducing the cost by modifying the architectural interoperability check by e.g. comparing each AEI of the loop with the overall loop interaction with that AEI excluded, otherwise the problem about deadlock freedom discussed in the first part of Sect. 4 shows up. It is however worth noting that resorting to the architectural interoperability check is more convenient than globally checking $I(C_1, \ldots, C_n)$ for deadlock. The first reason is that, while performing the former check, the state space of $I(C_1, \ldots, C_n)$ can be reduced via $\approx_{\mathrm{B}}$ thanks to the use of the hiding operator in Def. 5.1. The second reason is that the former check provides diagnostic information in case of mismatch. For instance, in the scenario of

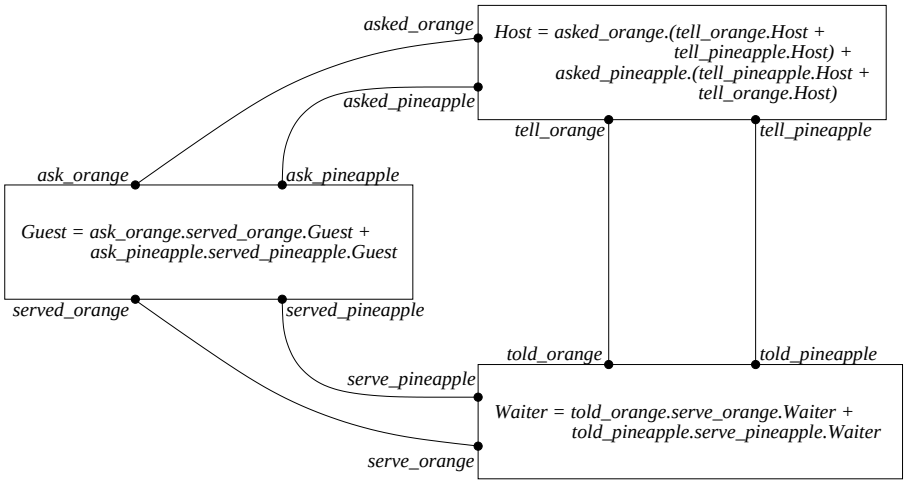

Figure 2. Flow graph representation of the guest analogy

Fig. 2, we find that $Guest$ fails the architectural interoperability check. By exploiting the modal logic based diagnostic information provided by tools like [4] in case of equivalence checking failure, we know that if $Guest$ performs e.g. ask_orange then it necessarily performs $served\_orange$ afterwards, while this is not the case for the whole loop. Since by direct inspection we see that $Guest$ is correct, we know that the cause of the mismatch is in the rest of the loop. In general, when a loop is formed by many AEIs, if one of such AEIs fails the architectural interoperability check, but by direct inspection we see that it and its adjacent AEIs are correct, the loop can be shrinked by replacing those AEIs with their parallel composition and the procedure is repeated until the cause of the architectural mismatch is identified.

6. Cruise Control System

In this section we assess the validity of our novel architectural interoperability check to discover a known architectural mismatch in the cruise control system examined in [9].

An automobile cruise control system is controlled by three buttons: on, off, and resume. When the engine is running and on is pressed, the cruise control system records the current speed and maintains the car at this speed. When the accelerator, brake, or off is pressed, the cruise control system disengages but retains the speed setting. If resume is pressed, the system accelerates or deaccelerates the car back to the previously recorded speed.

The PADL description of the system is depicted in Fig. 3. The behavior of each AEI is shown in Table 3. The system has 84 states and 230 transitions.

It can be globally verified by tools like [4] that this system is deadlock free, the reason being that speed measurement related interactions take place endlessly as long as the engine is running. Indeed, if we hide those interactions, the presence of deadlock shows up. The existence of a problem is revealed by the fact that, when considering the loop of Fig. 3, $Sensor$ fails the architectural interoperability check. By exploiting the modal logic based diagnostic information stemming from the failure of equivalence checking, we discover that the interaction of the loop in Fig. 3 deadlocks after performing $turn\_engine\_on$, $press\_on$, $turn\_engine\_off$, and $turn\_engine\_on$ again, while $Sensor$ does not, and that all we have to do in order to overcome this problem is to insert action $trigger\_disable\_speed\_control$ after $turned\_engine\_off$ in $Cruising$. It is worth observing that we have been able to automatically and systematically discover such a subtle mismatch thanks to our architectural interoperability check, whereas in [9] it is mentioned that the same error went originally undetected for some years and was discovered for the first time while animating the specification.

7. Conclusion

In this paper we have strengthened the architectural compatibility check of [3], thus subsuming the compatibility check of [2] and equalling the effectiveness of the algorithm of [8, 6] in a uniform framework thanks to standard observational equivalences, and we have introduced a novel architectural interoperability check, whose adequacy has been assessed on a cruise control system. Both checks, which rely on the properties of the observational equivalences defined for process algebras, come equipped with a deadlock freedom result, which makes them more convenient – for efficiency and/or diagnosis related reasons – than globally verifying deadlock freedom. A lesson learnt is that process algebras are well suited at the architectural level of design

$Sensor$	$\triangleq$	$turn\_engine\_on.Sensor'$
$Sensor'$	$\triangleq$	$press\_accelerator.Sensor' +$ $press\_brake.Sensor' +$ $press\_on.Sensor' +$ $press\_off.Sensor' +$ $press\_resume.Sensor' +$ $turn\_engine\_off.Sensor$
$CruiseController$	$\triangleq$	$Inactive$
$Inactive$	$\triangleq$	$turned\_engine\_on.trigger\_clear\_speed.Active$
$Active$	$\triangleq$	$pressed\_accelerator.Active +$ $pressed\_brake.Active +$ $pressed\_on.trigger\_enable\_speed\_control.trigger\_record\_speed.Cruising +$ $pressed\_off.Active +$ $pressed\_resume.Active +$ $turned\_engine\_off.Inactive$
$Cruising$	$\triangleq$	$pressed\_accelerator.trigger\_disable\_speed\_control.StandBy +$ $pressed\_brake.trigger\_disable\_speed\_control.StandBy +$ $pressed\_on.trigger\_enable\_speed\_control.trigger\_record\_speed.Cruising +$ $pressed\_off.trigger\_disable\_speed\_control.StandBy +$ $pressed\_resume.Cruising +$ $turned\_engine\_off.Inactive$
$StandBy$	$\triangleq$	$pressed\_accelerator.trigger\_disable\_speed\_control.StandBy +$ $pressed\_brake.trigger\_disable\_speed\_control.StandBy +$ $pressed\_on.trigger\_enable\_speed\_control.trigger\_record\_speed.Cruising +$ $pressed\_off.trigger\_disable\_speed\_control.StandBy +$ $pressed\_resume.trigger\_enable\_speed\_control.Cruising +$ $turned\_engine\_off.Inactive$
$SpeedDetector$	$\triangleq$	$turned\_engine\_on.WheelRevCounter$
$WheelRevCounter$	$\triangleq$	$measure\_speed.signal\_speed.WheelRevCounter +$ $turned\_engine\_off.SpeedDetector$
$SpeedController$	$\triangleq$	$Disabled$
$Disabled$	$\triangleq$	$signalled\_speed.Disabled +$ $triggered\_clear\_speed.clear\_speed.Disabled +$ $triggered\_enable\_speed\_control.enable\_speed\_control.Enabled +$ $triggered\_disable\_speed\_control.disable\_speed\_control.Disabled$
$Enabled$	$\triangleq$	$signalled\_speed.maintain\_speed.adjust\_throttle.Enabled +$ $triggered\_enable\_speed\_control.enable\_speed\_control.Enabled +$ $triggered\_record\_speed.record\_speed.Enabled +$ $triggered\_disable\_speed\_control.disable\_speed\_control.Disabled$
$Throttle$	$\triangleq$	$adjusted\_throttle.Throttle$

Table 3. Component behaviors for the cruise control system

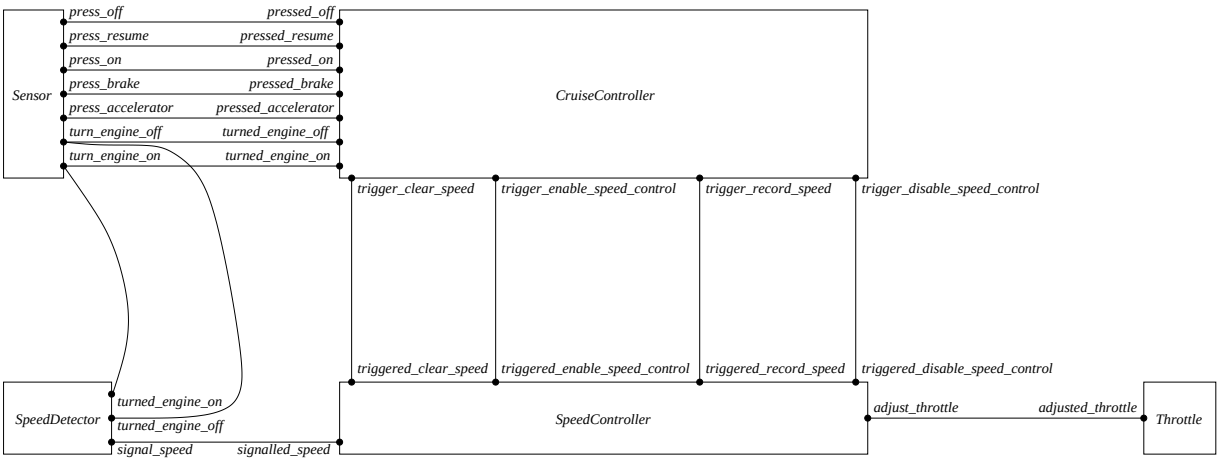

Figure 3. Flow graph representation of the cruise control system

not only because of their compositional modeling facilities, but also for their analysis related machinery that enables us to detect architectural mismatches in a rather efficient way and to get some diagnostic information useful for identifying the mismatch sources.

As far as future work is concerned, first we would like to understand to which extent the loop shrinking procedure sketched at the end of Sect. 5 can be automated (to reduce the architect intervention). Second, we would like to weaken the architectural compatibility and interoperability checks using other observational equivalences and investigate to which extent the deadlock freedom results are still valid. Different observational equivalences may be adequate to detect architectural mismatches that can be characterized by properties other than deadlock. Third, we would like to understand whether our deadlock freedom results, which are local to AEIs and loop interactions, can be generalized to the whole AT. In other words, we would like to see whether there are further causes of architectural mismatches, and whether they can be managed by local checks.

Acknowledgements

This research has been funded by Progetto MURST Cofinanziato SALADIN.

References

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

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

[3] M. Bernardo, P. Ciancarini, L. Donatiello, *"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, pp. 140-148, San Diego (CA), 2000

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

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

[6] P. Inverardi, S. Uchitel, *"Proving Deadlock Freedom in Component-Based Programming"*, in Proc. of the *4th Int. Conf. on Fundamental Approaches to Software Engineering (FASE '01)*, LNCS 2029:60-75, Genova (Italy), 2001

[7] P. Inverardi, A.L. Wolf, *"Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model"*, in IEEE Trans. on Software Engineering 21:373-386, 1995

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

[9] J. Kramer, J. Magee, *"Exposing the Skeleton in the Coordination Closet"*, in Proc. of the *2nd Int. Conf. on Coordination Languages and Models (COORDINATION '97)*, LNCS 1282:18-31, Berlin (Germany), 1997

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

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

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

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