Non-Synchronous Communications in Process Algebraic Architectural Description Languages

Marco Bernardo and Edoardo Bontà

Università di Urbino "Carlo Bo" – Italy
Istituto di Scienze e Tecnologie dell'Informazione

Abstract. Architectural description languages are a useful tool for modeling complex software systems at a high level of abstraction and, if based on formal methods, for enabling the early verification of various properties among which correct component coordination. This is the case with process algebraic architectural description languages, as they have been equipped with several techniques for verifying the absence of coordination mismatches in the case of synchronous communications. The objective of this paper is twofold. On the modeling side, we show how to enhance the expressiveness of a typical process algebraic architectural description language by including the capability of representing non-synchronous communications, in such a way that the usability of the original language is preserved. On the analysis side, we show how to modify the compatibility check for acyclic topologies and the interoperability check for cyclic topologies, in such a way that both checks can still be applied in the presence of non-synchronous communications.

1 Introduction

Several architectural description languages have been proposed in the literature. Many of them – like, e.g., Wright [2], Darwin/FSP [8], PADL [1], and π-ADL [10] – are based on process algebra [9] due to its support to compositional modeling. On the analysis side, process algebraic ADLs inherit all the techniques applicable to process algebra, like model checking and equivalence checking. In addition, such languages are equipped with ad-hoc analysis techniques (see, e.g., [2, 7, 4, 1]) mostly based on behavioral equivalences. These techniques are useful for (*i*) detecting coordination mismatches – deriving from components that are correct if taken separately but that do not satisfy certain requirements when assembled together – and (*ii*) generating diagnostic information – in order to pinpoint those components from which mismatches arise.

The ad-hoc analysis techniques proposed in the literature deal only with synchronous communications. In that setting, all ports of software components are blocking. A component waiting on a synchronous input port cannot proceed until an output is sent by another component. Similarly, a component issuing an output via a synchronous output port cannot proceed until another component is willing to receive. This is an important case, especially when verifying coordination properties like the deadlock freedom of software systems resulting from the assembly of individually deadlock-free components. However, in general software components can be involved not only in synchronous communications, but also in non-synchronous communications.

The first contribution of this paper is to show how to enhance the expressiveness of a typical process algebraic architectural description language by including the capability of representing non-synchronous communications, in such a way that the usability of the original language is preserved. More specifically, we focus on PADL [1] and we extend it by means of additional qualifiers useful to distinguish among synchronous, semi-synchronous, and asynchronous ports.

Semi-synchronous ports are not blocking. A semi-synchronous port of a component succeeds if there is another component ready to communicate with it, otherwise it raises an exception so as not to block the component to which it belongs. For example, a semi-synchronous input port can be used to model accesses to a tuple space via (non-blocking) input or read probes [6]. A semi-synchronous output port can instead be used to model the (non-blocking) interplay between a graphical user interface and an underlying application whenever the latter cannot do certain tasks requested by the user.

Analogously, asynchronous ports are not blocking. Here the reason is that the beginning and the end of the communications in which these ports are involved are completely decoupled. For instance, an asynchronous output port can be used to model output operations on a tuple space. An asynchronous input port can instead be used to model the periodical check for the presence of information received from an event notification service [5].

The semantic treatment of non-synchronous communications is completely transparent to PADL users. They only have to specify appropriate synchronicity-related qualifiers in their descriptions, hence the degree of usability of PADL is unaffected. We will see that semi-synchronous ports can easily be handled with suitable semantic rules generating exceptions whenever necessary, whereas asynchronous ports require the addition of implicit repository-like components.

The second contribution of this paper is to show how to modify the compatibility check for acyclic topologies and the interoperability check for cyclic topologies introduced in [1], in such a way that both checks can still be applied in the presence of non-synchronous communications.

This paper is organized as follows. In Sect. 2 we recall PADL. In Sect. 3 we extend its syntax with semi-synchronous and asynchronous ports and we consequently revise its semantics. In Sect. 4 we modify the architectural compatibility and interoperability checks in order to deal with non-synchronous communications as well. The modified checks are illustrated via an applet-based simulator for a cruise control system. In Sect. 5 we provide some concluding remarks.

2 The Architectural Description Language PADL

PADL [1] is a process algebraic architectural description language. In this section we present the syntax and the semantics for PADL after recalling some basic notions of process algebra.

2.1 Process Algebra

Process algebra [9] provides a set of operators by means of which the behavior of a system can be described in an action-based, compositional way. Given a set *Name* of action names including τ for invisible actions, we will consider a process algebra PA with the following process term syntax:

$P ::=$	$\underline{0}$	inactive process	
$\mid$	B	process constant	$(B \stackrel{\Delta}{=} P)$
$\mid$	$a.P$	action prefix	$(a \in \mathit{Name})$
$\mid$	$P + P$	alternative composition	
$\mid$	$P \parallel_S P$	parallel composition	$(S \subseteq \mathit{Name} - \{\tau\})$
$\mid$	P/H	hiding	$(H \subseteq \mathit{Name} - \{\tau\})$
$\mid$	$P[\varphi]$	relabeling	$(\varphi : \mathit{Name} \to \mathit{Name},\ \varphi^{-1}(\tau) = \{\tau\})$

Operational semantic rules map every closed and guarded process term P of PA to a state-transition graph $[\![P]\!]$ called labeled transition system, where each state corresponds to a process term derivable from P, the initial state corresponds to P, and each transition is labeled with the corresponding action.

Process terms are compared and manipulated by means of behavioral equivalences. Among the various approaches, for PA we consider weak bisimilarity, according to which two process terms are equivalent if they are able to mimic each other's visible behavior stepwise.

A symmetric relation $\mathcal{R}$ is a weak bisimulation if for all $(P_1, P_2) \in \mathcal{R}$ and $a \in \mathit{Name} - \{\tau\}$: *(i)* whenever $P_1 \xrightarrow{a} P'_1$, then $P_2 \xRightarrow{\tau^* a \tau^*} P'_2$ and $(P'_1, P'_2) \in \mathcal{R}$; *(ii)* whenever $P_1 \xrightarrow{\tau} P'_1$, then $P_2 \xRightarrow{\tau^*} P'_2$ and $(P'_1, P'_2) \in \mathcal{R}$. Weak bisimilarity $\approx_{\mathrm{B}}$ is the union of all the weak bisimulations.

2.2 PADL Textual and Graphical Notations

A PADL description represents an architectural type, which is a family of software systems sharing certain constraints on the observable behavior of their components as well as on their topology.

The textual description of an architectural type starts with the name and the formal parameters (initialized with default values) of the architectural type. The available data types are boolean, integer, real, list, array, record, and generic object. The textual description then comprises two sections.

The first section defines the behavior of the system family by means of types of software components and connectors, which are collectively called architectural element types. The definition of an AET starts with its name and its formal parameters and consists of the specification of its behavior and its interactions.

The behavior of an AET has to be provided in the form of a sequence of defining equations written in a verbose variant of PA allowing only for the inactive process (rendered as `stop`), the value-passing action prefix operator with a possible boolean guard condition, the alternative composition operator (rendered as `choice`), and recursion.

The interactions are those actions occurring in the process algebraic specification of the behavior that act as interfaces for the AET, while all the other actions are assumed to represent internal activities. Each interaction has to be equipped with two qualifiers. The first qualifier establishes whether the interaction is an input or output interaction.

The second qualifier describes the multiplicity of the communications in which the interaction can be involved. We distinguish among uni-interactions mainly involved in one-to-one communications (qualifier `UNI`), and-interactions guiding inclusive one-to-many communications (qualifier `AND`), or-interactions guiding selective one-to-many communications (qualifier `OR`). It can also be established that an output or-interaction depends on an input or-interaction, in order to guarantee that a selective one-to-many output is sent to the same element from which a selective many-to-one input was received (keyword `DEP`).

The second section of the PADL description defines the topology of the system family. This is accomplished in three steps. First we have the declaration of the instances of the AETs – called AEIs – which represent the actual system components and connectors, together with their actual parameters. Then we have the declaration of the architectural (as opposed to local) interactions, which are some of the interactions of the AEIs that act as interfaces for the whole systems of the family. Finally, we have the declaration of the architectural attachments among the local interactions of the AEIs, which make the AEIs communicate with each other. An attachment is admissible only if it goes from an output interaction of an AEI to an input interaction of another AEI. Moreover, a uni-interaction can be attached to only one interaction, whereas an and-/or-interaction can be attached to (several) uni-interactions only.

```
ARCHI_TYPE                    ◁name and initialized formal parameters▷

  ARCHI_BEHAVIOR
        ⋮                       ⋮
    ARCHI_ELEM_TYPE           ◁AET name and formal parameters▷
      BEHAVIOR                ◁sequence of PA defining equations built from
                               stop, action prefix, choice, and recursion▷
      INPUT_INTERACTIONS      ◁input uni/and/or-interactions▷
      OUTPUT_INTERACTIONS     ◁output uni/and/or-interactions▷
        ⋮                       ⋮

  ARCHI_TOPOLOGY
    ARCHI_ELEM_INSTANCES      ◁AEI names and actual parameters▷
    ARCHI_INTERACTIONS        ◁architecture-level AEI interactions▷
    ARCHI_ATTACHMENTS         ◁attachments between AEI local interactions▷

END
```

Besides the textual notation, PADL comes equipped with a graphical notation that is an extension of the flow graph notation [9]. In an enriched flow graph, AEIs are depicted as boxes, local (resp. architectural) interactions are depicted as small black circles (resp. white squares) on the box border, and attachments are depicted as directed edges between pairs each composed of a local output interaction and a local input interaction. The small circle/square of an interaction is extended with a triangle (resp. bisected triangle) outside the AEI box if the interaction is an and-interaction (resp. or-interaction).

Example 1. Suppose we need to model a scenario in which there is a server that can be contacted at any time by two identically behaving clients. Assume that the server has no buffer for holding incoming requests and that, after sending a request, a client cannot proceed until it receives a response from the server. Since the behavior of the two clients is identical, a single client AET suffices:

```
ARCHI_ELEM_TYPE Client_Type(void)
  BEHAVIOR
    Client(void; void) =
      process . send_request . receive_response . Client()
  INPUT_INTERACTIONS  UNI receive_response
  OUTPUT_INTERACTIONS UNI send_request
```

where `process` is an internal action. The server AET can be defined as follows:

```
ARCHI_ELEM_TYPE Server_Type(void)
  BEHAVIOR
    Server(void; void) =
      receive_request . compute_response . send_response . Server()
  INPUT_INTERACTIONS  OR receive_request
  OUTPUT_INTERACTIONS OR send_response   DEP receive_request
```

where `compute_response` is an internal action, while `send_response` is declared to depend on `receive_request` in order to make sure that each response is sent back to the client that issued the corresponding request. Finally, we declare the topology of the system as follows:

```
ARCHI_TOPOLOGY
  ARCHI_ELEM_INSTANCES
    C_1 : Client_Type();
    C_2 : Client_Type();
    S   : Server_Type()
  ARCHI_INTERACTIONS
    void
  ARCHI_ATTACHMENTS
    FROM C_1.send_request TO S.receive_request;
    FROM C_2.send_request TO S.receive_request;
    FROM S.send_response  TO C_1.receive_response;
    FROM S.send_response  TO C_2.receive_response
```

where the dot notation has to be used so as to avoid ambiguities in cases in which the same action name denotes interactions belonging to different AEIs. ■

2.3 The Semantics for PADL

The semantics for PADL is given by translation into PA. The meaning of a PADL description is a process term stemming from the parallel composition of the process algebraic specifications of the behavior of the AEIs declared in the description, with synchronization sets being determined by attachments.

Let $\mathcal{C}$ be an AET with formal parameters $fp_1, \ldots, fp_m$ and behavior given by the sequence $\mathcal{E}$ of defining equations. Let C be an AEI of type $\mathcal{C}$ with actual parameters $ap_1, \ldots, ap_m$. Then the semantics of C is defined as follows:

$$\boxed{[\![C]\!] \;=\; \textit{or-rewrite}(\mathcal{E}\{ap_1/fp_1, \ldots, ap_m/fp_m\})}$$

where $\{\_/\_, \ldots, \_/\_\}$ denotes a syntactical substitution, while function *or-rewrite* inductively rewrites the body of any defining equation of $\mathcal{E}$ by replacing each occurrence of any or-interaction with fresh uni-interactions. More precisely, if or-interaction a of C is involved in $\textit{attach-no}(C.a) = l \geq 2$ attachments, then:

$$\boxed{\begin{aligned} \textit{or-rewrite}(a.P) \;=\; \texttt{choice}\{&a_1.\textit{or-rewrite}(P),\\ &\vdots\\ &a_l.\textit{or-rewrite}(P)\} \end{aligned}}$$

Consider now a set $\{C_1, \ldots, C_n\}$ of AEIs and let us denote by $\mathcal{LI}_{C_j}$ the set of local interactions of C_j and by $\mathcal{LI}_{C_j;C_1,\ldots,C_n} \subseteq \mathcal{LI}_{C_j}$ the set of local interactions of C_j attached to $\{C_1, \ldots, C_n\}$. In order to make such AEIs interact in the framework of PA – where only actions with the same name can synchronize – we need a set $\mathcal{S}(C_1, \ldots, C_n)$ of fresh action names, one for each pair of attached local uni-interactions in $\{C_1, \ldots, C_n\}$ and for each set of local uni-interactions attached to the same local and-interaction in $\{C_1, \ldots, C_n\}$.

Then we need suitable injective relabeling functions $\varphi_{C_j;C_1,\ldots,C_n}$ mapping each $\mathcal{LI}_{C_j;C_1,\ldots,C_n}$ to $\mathcal{S}(C_1, \ldots, C_n)$ in such a way that:

$$\boxed{\varphi_{C_j;C_1,\ldots,C_n}(a_1) \;=\; \varphi_{C_g;C_1,\ldots,C_n}(a_2)}$$

if and only if $C_j.a_1$ and $C_g.a_2$ are attached to each other or to the same and-interaction. To ensure renaming uniqueness, $\mathcal{S}(C_1, \ldots, C_n)$ can be built by concatenating the original names of attached interactions – e.g., $C_j.a_1\#C_g.a_2$.

The interacting semantics of C_j with respect to $\{C_1, \ldots, C_n\}$ is defined as follows:

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

In general, the interacting semantics of $\{C'_1, \ldots, C'_{n'}\} \subseteq \{C_1, \ldots, C_n\}$ with respect to $\{C_1, \ldots, C_n\}$ is defined as follows:

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

where $\mathcal{S}(C'_j, C'_g; C_1, \dots, C_n) = \mathcal{S}(C'_j; C_1, \dots, C_n) \cap \mathcal{S}(C'_g; C_1, \dots, C_n)$ is the pairwise synchronization set of C'_j and C'_g with respect to $\{C_1, \dots, C_n\}$, with $\mathcal{S}(C'_j; C_1, \dots, C_n) = \varphi_{C'_j; C_1, \dots, C_n}(\mathcal{LI}_{C'_j; C_1, \dots, C_n})$ being the synchronization set of C'_j with respect to $\{C_1, \dots, C_n\}$. Finally, the semantics of an architectural type $\mathcal{A}$ formed by the set of AEIs $\{C_1, \dots, C_n\}$ is defined as follows:

$$\boxed{[\![\mathcal{A}]\!] \; = \; [\![C_1, \dots, C_n]\!]_{C_1, \dots, C_n}}$$

Example 2. Consider the client-server system described in Ex. 1. Then $[\![\texttt{C\_1}]\!]$ and $[\![\texttt{C\_2}]\!]$ coincide with the defining equation for `Client`, whereas $[\![\texttt{S}]\!]$ is given by the following defining equation obtained from the one for `Server` after manipulating the occurring or-interactions:

```
Server'(void; void) =
  choice
  {
    receive_request_1 . compute_response . send_response_1 . Server'(),
    receive_request_2 . compute_response . send_response_2 . Server'()
  }
```

The semantics of the whole description is given by the following process term:

$$\begin{array}{l}
[\![\texttt{C\_1}]\!][\texttt{send\_request} \mapsto \texttt{C\_1.send\_request\#S.receive\_request\_1}, \\
\quad\quad \texttt{receive\_response} \mapsto \texttt{S.send\_response\_1\#C\_1.receive\_response}] \\
\quad\quad\quad \|_{\emptyset} \\
\quad [\![\texttt{C\_2}]\!][\texttt{send\_request} \mapsto \texttt{C\_2.send\_request\#S.receive\_request\_2}, \\
\quad\quad\quad \texttt{receive\_response} \mapsto \texttt{S.send\_response\_2\#C\_2.receive\_response}] \\
\quad\quad\quad\quad \|_{\{\texttt{C\_1.send\_request\#S.receive\_request\_1}, \\
\quad\quad\quad\quad\quad \texttt{S.send\_response\_1\#C\_1.receive\_response}, \\
\quad\quad\quad\quad\quad \texttt{C\_2.send\_request\#S.receive\_request\_2}, \\
\quad\quad\quad\quad\quad \texttt{S.send\_response\_2\#C\_2.receive\_response}\}} \\
\quad\quad [\![\texttt{S}]\!][\texttt{receive\_request\_1} \mapsto \texttt{C\_1.send\_request\#S.receive\_request\_1}, \\
\quad\quad\quad \texttt{send\_response\_1} \mapsto \texttt{S.send\_response\_1\#C\_1.receive\_response}, \\
\quad\quad\quad \texttt{receive\_request\_2} \mapsto \texttt{C\_2.send\_request\#S.receive\_request\_2}, \\
\quad\quad\quad \texttt{send\_response\_2} \mapsto \texttt{S.send\_response\_2\#C\_2.receive\_response}] \; \blacksquare
\end{array}$$

3 Semi-Synchronous and Asynchronous Interactions

All the interactions occurring in a PADL description can be involved only in synchronous communications, thus causing input interactions and output interactions to be blocking operations. In order to increase the expressiveness of PADL, within the interface of each AET we will provide support for distinguishing among synchronous, semi-synchronous and asynchronous interactions. The usability of the language will be preserved by means of suitable synchronicity-related qualifiers that are made available to PADL users.

In this section we enrich the textual and graphical notations in order to express non-synchronous interactions, then we revise the semantics accordingly. The nine resulting forms of communication are summarized by Fig. 1, with the first one being the only one originally available in PADL.

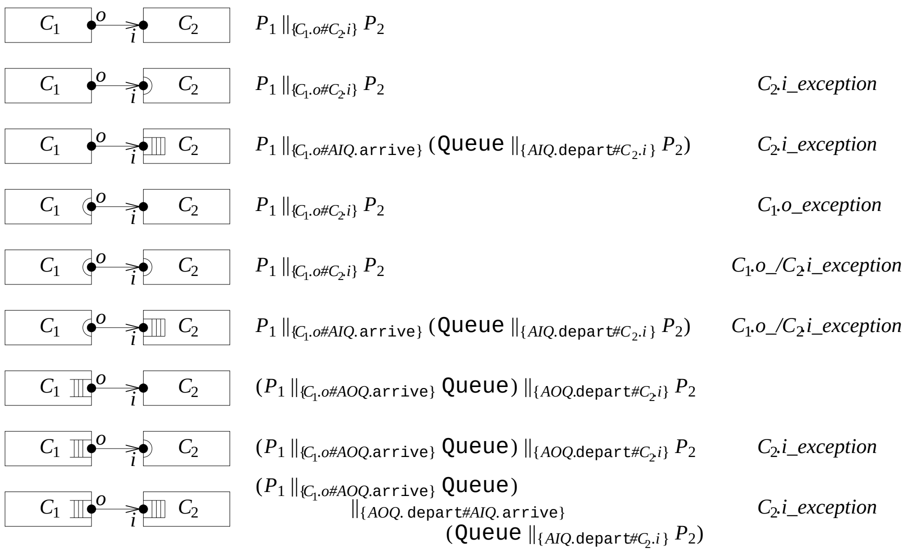
C1 o i C2 P1 ||{C1.o#C2.i} P2
C1 o i C2 P1 ||{C1.o#C2.i} P2 C2.i_exception
C1 o i C2 P1 ||{C1.o#AIQ.arrive} (Queue ||{AIQ.depart#C2.i} P2) C2.i_exception
C1 o i C2 P1 ||{C1.o#C2.i} P2 C1.o_exception
C1 o i C2 P1 ||{C1.o#C2.i} P2 C1.o_/C2.i_exception
C1 o i C2 P1 ||{C1.o#AIQ.arrive} (Queue ||{AIQ.depart#C2.i} P2) C1.o_/C2.i_exception
C1 o i C2 (P1 ||{C1.o#AOQ.arrive} Queue) ||{AOQ.depart#C2.i} P2
C1 o i C2 (P1 ||{C1.o#AOQ.arrive} Queue) ||{AOQ.depart#C2.i} P2 C2.i_exception
C1 o i C2 (P1 ||{C1.o#AOQ.arrive} Queue) ||{AOQ.depart#AIQ.arrive} (Queue ||{AIQ.depart#C2.i} P2) C2.i_exception

Fig. 1. Synchronous, semi-synchronous and asynchronous communications

3.1 Enriching PADL Textual and Graphical Notations

In the textual notation of PADL we introduce a third qualifier for interactions, to be used in the definition of the AETs. Such a qualifier establishes whether an interaction is synchronous, semi-synchronous, or asynchronous. The related keywords are `SYNC` (default value), `SSYNC`, and `ASYNC`, respectively.

While a synchronous interaction blocks the AEI executing it as long as the interactions attached to it are not ready, this is not the case with non-synchronous interactions. More precisely, a semi-synchronous interaction raises an exception if it cannot take place immediately due to the (temporary or permanent) unavailability of the interactions attached to it, so that the AEI executing it can proceed anyway. Likewise, in the case of an asynchronous interaction the beginning and the end of the communication are decoupled, hence the AEI executing the interaction will never block.

A boolean variable s.`success` is associated with each semi-synchronous interaction s. This implicitly declared variable is made available to PADL users in order to catch exceptions. In this way situations in which different behaviors have to be undertaken depending on the outcome of s can easily be managed.

In the graphical notation a semi-synchronous interaction is depicted by extending the small circle/square of the interaction with an arc inside the AEI box. An asynchronous interaction, instead, is depicted by extending the small circle/square with a buffer inside the AEI box.

Example 3. Consider again the client-server system described in Ex. 1. Since the server has no buffer for incoming requests, each client may want to send a request only if the server is not busy, so that the client can keep working instead of passively waiting for the server to become available. This can be described by transforming send_request into a semi-synchronous interaction:

```
ARCHI_ELEM_TYPE Client_Type(void)
  BEHAVIOR
    Client(void; void) =
      process . send_request .
        choice
        {
          cond(send_request.success = true) ->
                                        receive_response . Client(),
          cond(send_request.success = false) ->
                                        keep_processing . Client()
        }
  INPUT_INTERACTIONS  SYNC  UNI receive_response
  OUTPUT_INTERACTIONS SSYNC UNI send_request
```

On the other hand, the server should not make any assumption about the status of its clients, as these may be much more complicated than the description above. In particular, when sending out a response to a client, the server should not be blocked by the temporary or permanent unavailability of that client, as this would decrease the quality of service. This can be achieved by using keyword ASYNC in the declaration of interaction send_response within Server_Type. ■

3.2 Semantics of Semi-Synchronous Interactions: Additional Rules

A semi-synchronous interaction s executed by an AEI C gives rise to a transition labeled with s within $[\![C]\!]$. However, in an interacting context this transition has to be relabeled with *s_exception* if s cannot immediately participate in a communication. This is accomplished by means of additional semantic rules.

Suppose that the output interaction o of an AEI C_1 is attached to the input interaction i of an AEI C_2. Let $C_1.o\#C_2.i$ be the fresh action name associated with their synchronization, P_1 (resp. P_2) be the process term representing the current state of $[\![C_1]\!]_{C_1,C_2}$ (resp. $[\![C_2]\!]_{C_1,C_2}$), and $S = \mathcal{S}(C_1, C_2; C_1, C_2)$.

If o is synchronous and i is semi-synchronous – which is the second form of communication depicted in Fig. 1 – then the following additional semantic rule is necessary for handling exceptions:

$$\boxed{\dfrac{P_1 \xrightarrow{C_1.o\#C_2.i}\!\!\!\!\!\!\!\!\!\!\!\!/\;\;\;\;\;\;\;\;\;\; \qquad P_2 \xrightarrow{C_1.o\#C_2.i} P_2'}{P_1 \parallel_S P_2 \xrightarrow{C_2.i\_exception} P_1 \parallel_S P_2' \qquad C_2.i.\texttt{success} = \texttt{false}}}$$

In the symmetric case in which o is semi-synchronous and i is synchronous – which corresponds to the fourth form of communication depicted in Fig. 1 – the following additional semantic rule is necessary for handling exceptions:

$$\boxed{\dfrac{P_1 \xrightarrow{C_1.o\#C_2.i} P_1' \qquad P_2 \xrightarrow{C_1.o\#C_2.i}\!\!\!\!\!\!\!\!\!\!\!\!/\;\;\;\;\;\;\;\;\;\;}{P_1 \parallel_S P_2 \xrightarrow{C_1.o\_exception} P_1' \parallel_S P_2 \qquad C_1.o.\texttt{success} = \texttt{false}}}$$

Finally, in the case in which both o and i are semi-synchronous – which corresponds to the fifth form of communication depicted in Fig. 1 – we have the previous two semantic rules together.

3.3 Semantics of Asynchronous Interactions: Implicit AEIs

While semi-synchronous interactions are dealt with by means of suitable semantic rules accounting for possible exceptions, asynchronous interactions need a different treatment because of the decoupling between the beginning and the end of the communications in which those interactions are involved.

After the or-rewriting process, for each asynchronous uni-/and-interaction we have to introduce an additional implicit AEI that behaves as an unbounded buffer, as shown in the third, sixth, seventh, eighth and ninth form of communication depicted in Fig. 1. This AEI is of the following type, where `arrive` is an always-available synchronous interaction, whereas `depart` is a synchronous interaction enabled only if the buffer is not empty.

```
ARCHI_ELEM_TYPE Async_Queue(void)
  BEHAVIOR
    Queue(int i := 0; void) =
      choice
      {
        cond(true)  -> arrive . Queue(i + 1),
        cond(i > 0) -> depart . Queue(i - 1)
      }
  INPUT_INTERACTIONS  SYNC --- arrive
  OUTPUT_INTERACTIONS SYNC --- depart
```

In the case of an asynchronous output interaction o, this is implicitly converted into a synchronous uni-interaction and attached to `arrive`, which is declared as a uni-interaction. By contrast, `depart`, which is declared as a uni-/and-interaction depending on whether o was a uni-/and-interaction, is attached to the input interactions originally attached to o.

In the case of an asynchronous input interaction i, `depart` is declared as a uni-interaction and implicitly attached to i, which is implicitly converted into a semi-synchronous uni-interaction. By contrast, the output interactions originally attached to i are attached to `arrive`, which is declared as a uni-/and-interaction depending on whether i was a uni-/and-interaction.

Note that i becomes semi-synchronous because the communications between `depart` and i must not block the AEI executing i whenever the buffer is empty. Thus i is subject to the first additional semantic rule defined in Sect. 3.2.

3.4 Revising PADL Semantics

Due to the way non-synchronous interactions have been handled, we only need to revise the definition of the semantics of an AEI in isolation, while all the subsequent definitions given in Sect. 2.3 are unchanged. More precisely, we only have to take into account the possible presence of additional implicit AEIs acting as unbounded buffers for asynchronous interactions.

Suppose that AEI C has $h \in \mathbb{N}_{>0}$ asynchronous input interactions $i_1, \dots, i_h$ – handled through the related additional implicit AEIs $AIQ_1, \dots, AIQ_h$ – and $k \in \mathbb{N}_{>0}$ asynchronous output interactions $o_1, \dots, o_k$ – handled through the related additional implicit AEIs $AOQ_1, \dots, AOQ_k$. Then $[\![C]\!]$ is defined as follows:

$$\boxed{\begin{array}{l}((\overbrace{\texttt{Queue} \parallel_{\emptyset} \dots \parallel_{\emptyset} \texttt{Queue}}^{h})\,[\varphi_{C,\text{async}}]) \parallel_{\{AIQ_1.depart\#C.i_1, \dots, AIQ_h.depart\#C.i_h\}} \\ \quad \mathit{or\text{-}rewrite}(\mathcal{E}\{ap_1/fp_1, \dots, ap_m/fp_m\})\,[\varphi_{C,\text{async}}] \\ \qquad \parallel_{\{C.o_1\#AOQ_1.arrive, \dots, C.o_k\#AOQ_k.arrive\}} ((\underbrace{\texttt{Queue} \parallel_{\emptyset} \dots \parallel_{\emptyset} \texttt{Queue}}_{k})\,[\varphi_{C,\text{async}}])\end{array}}$$

where $\varphi_{C,\text{async}}$ transforms $C.i_1, \dots, C.i_h, C.o_1, \dots, C.o_k$ and the related attached interactions of $AIQ_1, \dots, AIQ_h, AOQ_1, \dots, AOQ_k$ into the corresponding fresh names occurring in the two synchronization sets.

4 Modifying Architectural Checks

The objective of the architectural checks developed in [1] is to infer certain architectural properties – which thus involve only interactions – from the properties of the individual AEIs through a topological reduction process based on equivalence checking. In case of failure, such checks provide diagnostic information useful to single out components responsible for possible property violations.

The starting point in [1] is given by abstract variants of enriched flow graphs, where vertices correspond to AEIs and two vertices are linked by an edge if and only if attachments have been declared among their interactions. These graphs are arbitrary combinations of stars and cycles, which are thus viewed as basic topological formats.

The strategy proposed in [1] is to reduce the whole topology of an architectural type to a single equivalent AEI that satisfies the properties of interest. This is accomplished by applying specific checks locally to stars and cycles occurring in the abstract enriched flow graph of the architectural type. If passed, each check allows the star/cycle in which it has been employed to be replaced by an equivalent AEI in the star/cycle itself that satisfies the properties of interest.

In this section we show how to modify the compatibility check for stars and the interoperability check for cycles, in such a way that both checks can still be applied in the presence of non-synchronous interactions. Although these checks are conceived for an entire class of properties [1], for the sake of simplicity here the considered property is deadlock freedom and the behavioral equivalence chosen among those preserving deadlock freedom is weak bisimilarity $\approx_{\text{B}}$ (Sect. 2.1).

4.1 Revising Closed Interacting Semantics

The considered architectural checks must be applied to closed variants of the interacting semantics of AEIs, where all the internal actions are hidden. In the framework of PADL enriched with non-synchronous interactions, also the asynchronous interactions have to be hidden together with the interactions of the related additional implicit AEIs to which they are re-attached. The reason is that all of those interactions cannot communicate with the rest of the system, hence they cannot affect architectural properties.

Let $\{C_1, \dots, C_n\}$ be a set of AEIs and let C_j be one of its AEIs having $h \in \mathbb{N}_{>0}$ asynchronous input interactions $i_1, \dots, i_h$ and $k \in \mathbb{N}_{>0}$ asynchronous output interactions $o_1, \dots, o_k$. The closed interacting semantics of C_j with respect to $\{C_1, \dots, C_n\}$ is defined as follows:

$$\boxed{\begin{aligned} [\![C_j]\!]^{\mathrm{c}}_{C_1,\dots,C_n} = \; & [\![C_j]\!]_{C_1,\dots,C_n} \,/\, (\mathit{Name} - \mathcal{LI}_{C_j;C_1,\dots,C_n}) \\ & /\, \{AIQ_1.\mathit{depart}\#C_j.i_1, \dots, AIQ_h.\mathit{depart}\#C_j.i_h, \\ & \quad C_j.o_1\#AOQ_1.\mathit{arrive}, \dots, C_j.o_k\#AOQ_k.\mathit{arrive}, \\ & \quad C_j.i_1\_\mathit{exception}, \dots, C_j.i_h\_\mathit{exception}\} \end{aligned}}$$

The closed interacting semantics $[\![C'_1, \dots, C'_{n'}]\!]^{\mathrm{c}}_{C_1,\dots,C_n}$ and the closed semantics $[\![\mathcal{A}]\!]^{\mathrm{c}}$ are defined accordingly.

4.2 Adapting Architectural Compatibility

A star is an acyclic portion of the abstract enriched flow graph of an architectural type, which is formed by a central AEI K and a border $\mathcal{B}_K = \{C_1, \dots, C_n\}$ including all the AEIs attached to K. In order to achieve a correct coordination between K and each $C_j \in \mathcal{B}_K$, the actual observable behavior of each C_j should coincide with the one expected by K. In other words, the observable behavior of K should not be altered by the addition of C_j to the border of the star.

Definition 1. *We say that K is compatible with $C_j \in \mathcal{B}_K$ iff:*

$$\boxed{([\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{\mathcal{S}(K,C_j;K,\mathcal{B}_K)} [\![C_j]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}) \,/\, H_j \;\approx_{\mathrm{B}}\; [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H_j}$$

where the hiding set H_j includes all the semi-synchronous interactions involved in attachments between K and C_j together with the related exceptions, as well as all the interactions of implicit AEIs associated with K (resp. C_j) that are attached to interactions of C_j (resp. K). ■

Note that $H_j = \emptyset$ whenever neither K nor C_j has semi-synchronous or asynchronous interactions. In fact, the presence of H_j is the novelty with respect to the definition of compatibility given in [1].

The reason why it makes sense to hide those semi-synchronous and asynchronous interactions is that they are not blocking, hence similarly to internal actions they cannot negatively affect component coordination.

The reason why it is necessary to hide each of them is that, within an AEI executing one of them the interaction takes place at a specific point with a specific outcome, while in the parallel composition of that AEI with other AEIs the same interaction can have a different outcome (semi-synchronous case) or can be delayed (asynchronous case). This may lead to detect inequivalence between the behavior of the individual AEI and the behavior of a set of AEIs including it – a compatibility violation – even in the absence of a real coordination mismatch.

We now extend the compatibility theorem of [1] to non-synchronous interactions. The additional constraint to satisfy is that no and-interaction occurring in the star can be non-synchronous or attached to a non-synchronous interaction.

Theorem 1. *Let $H = H_1 \cup \ldots \cup H_n$. Whenever $[\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H$ is deadlock free and K is compatible with any $C_j \in \mathcal{B}_K$, then the whole star $[\![K, \mathcal{B}_K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H$ is deadlock free provided that $H_j \cap H_g = \emptyset$ for all $j \neq g$.* ■

4.3 Adapting Architectural Interoperability

Consider a cycle $\{C_1, \ldots, C_n\}$ in the abstract enriched flow graph of an architectural type $\mathcal{A}$. As shown in [1], compatibility is not enough to deal with it. The reason is that the AEIs in the cycle can no longer be considered two-by-two, because each of them may interfere with any of the others. In order to achieve a correct coordination between any C_j and the rest of the cycle, the actual observable behavior of C_j should coincide with the one expected by the rest of the cycle. In other words, the observable behavior of the rest of the cycle should not be altered by the addition of C_j to the cycle.

Definition 2. *We say that C_j interoperates with the rest of the cycle iff:*

$$\boxed{[\![C_1, \ldots, C_n]\!]^{\mathrm{c}}_{\mathcal{A}} \,/\, (\mathit{Name} - \mathcal{S}(C_j; \mathcal{A})) \,/\, H_j \;\approx_{\mathrm{B}}\; [\![C_j]\!]^{\mathrm{c}}_{\mathcal{A}} \,/\, H_j}$$

where H_j includes all the semi-synchronous interactions involved in attachments between C_j and the rest of the cycle together with the related exceptions, as well as all the interactions of implicit AEIs associated with C_j (resp. the rest of the cycle) that are attached to interactions of the rest of the cycle (resp. C_j). ■

As in Sect. 4.2, the presence of H_j is the novelty with respect to the definition of interoperability given in [1]. We now extend the interoperability theorem of [1] to non-synchronous interactions.

Theorem 2. *Whenever there exists C_j in the cycle such that $[\![C_j]\!]^{\mathrm{c}}_{\mathcal{A}} \,/\, H_j$ is deadlock free and C_j interoperates with the rest of the cycle, then the whole cycle $[\![C_1, \ldots, C_n]\!]^{\mathrm{c}}_{\mathcal{A}} \,/\, (\mathit{Name} - \mathcal{S}(C_j; \mathcal{A})) \,/\, H_j$ is deadlock free.* ■

4.4 Example: An Applet-Based Simulator

In this section we discuss the application of the modified architectural checks by revisiting the cruise control system considered in [8, 3].

This system is governed by two pedals – accelerator and brake – and three buttons – on, off, and resume. When on is pressed, the cruise control system records the current speed and maintains the automobile at that speed. When the accelerator, the brake, or off is pressed, the cruise control system disengages but retains the speed setting. If resume is pressed later on, then the system is able to accelerate or decelerate the automobile to the previously recorded speed.

The cruise control system is formed by four software components: a sensor, a speed controller, a speed detector, and a speed actuator. The sensor detects the driver commands and forwards them to the speed controller, which in turn triggers the speed actuator. The speed detector periodically measures the number of wheel revolutions per time unit. The speed actuator adjusts the throttle on the basis of the triggers received from the controller and of the speed measured by the detector.

As an example, we report the definition of the sensor AET:

```
ARCHI_ELEM_TYPE Sensor_Type(void)
  BEHAVIOR
    Sensor_Off(void; void) =
      detected_engine_on . turn_engine_on . Sensor_On();
    Sensor_On(void; void) =
      choice
      {
        detected_accelerator . press_accelerator . Sensor_On(),
        detected_brake . press_brake . Sensor_On(),
        detected_on . press_on . Sensor_On(),
        detected_off . press_off . Sensor_On(),
        detected_resume . press_resume . Sensor_On(),
        detected_engine_off . turn_engine_off . Sensor_Off()
      }
  INPUT_INTERACTIONS  UNI detected_engine_on; detected_engine_off;
                          detected_accelerator; detected_brake;
                          detected_on; detected_off; detected_resume
  OUTPUT_INTERACTIONS UNI press_accelerator; press_brake;
                          press_on; press_off; press_resume
                      AND turn_engine_on; turn_engine_off
```

Suppose we want to design an applet-based simulator for such a system. The applet will have seven software buttons – corresponding to turning the engine on/off, the two pedals, and the three hardware buttons – together with a text area showing the sequence of buttons that have been pressed. When pressing one of the seven software buttons, the corresponding operation either succeeds or fails. In the first case, the applet can interact with the sensor and the text area is updated accordingly. In the second case – think, e.g., of pressing the accelerator button when the engine is off – the applet cannot interact with the sensor, rather it emits a beep.

In order not to block the simulator in case of failure, we need to model several operations of the applet through semi-synchronous interactions, as shown below:

```
ARCHI_ELEM_TYPE Applet_Type(void)
  BEHAVIOR
    Unallocated(void; void) =
      create_applet . start_applet . Active();
    Active(void; void) =
      choice
      {
        signal_engine_on . Checking(signal_engine_on.success),
        signal_accelerator . Checking(signal_accelerator.success),
        signal_brake . Checking(signal_brake.success),
        signal_on . Checking(signal_on.success),
        signal_off . Checking(signal_off.success),
        signal_resume . Checking(signal_resume.success),
        signal_engine_off . Checking(signal_engine_off.success),
        stop_applet . Inactive()
      };
    Checking(boolean success; void) =
      choice
      {
        cond(success = true)  -> update . Active(),
        cond(success = false) -> beep . Active(),
      };
    Inactive(void; void) =
      choice
      {
        start_applet . Active(),
        destroy_applet . Unallocated()
      }
  INPUT_INTERACTIONS  SYNC  UNI create_applet; destroy_applet;
                              start_applet; stop_applet
  OUTPUT_INTERACTIONS SSYNC UNI signal_engine_on; signal_engine_off;
                              signal_accelerator; signal_brake;
                              signal_on; signal_off; signal_resume
```

As far as the instance of `Applet_Type` is concerned, its four input interactions are related to user commands for starting/stopping the simulator. By contrast, its seven `signal_` output interactions are related to user commands for the cruise control system, hence they are attached to the corresponding `detected_` input interactions of the instance of `Sensor_Type`.

Suppose we wish to verify whether the applet-based simulator is deadlock free. From the topological viewpoint, the system is a cycle formed by four AEIs (sensor, controller, actuator, detector) with an additional AEI (applet) attached to one of them (sensor). Within the cycle there are no non-synchronous interactions, hence applying the modified interoperability check to the cycle boils down to applying the original check. The outcome is thus known from [3]: the cycle is deadlock free, because it is weakly bisimilar to the sensor and the sensor is deadlock free (Thm. 2).

Now the sensor and the applet constitute a degenerate star, for which the original compatibility check is not appropriate due to the presence of semi-synchronous interactions within the applet. By applying the modified architectural compatibility check, we see that the parallel composition of the closed interacting semantics of the applet and of the sensor is weakly bisimilar to the closed interacting semantics of the applet, where all the semi-synchronous interactions and the related exceptions have been hidden. Since the applet is deadlock free, we can conclude that the applet-based simulator is deadlock free (Thm. 1).

5 Conclusion

In this paper we have extended process algebraic ADLs by including semi-synchronous interactions – handled by means of suitable semantic rules – and asynchronous interactions – managed by adding implicit buffer-like components. Besides enhancing the expressiveness of a typical process algebraic ADL without compromising its usability, we have shown that architectural checks for acyclic and cyclic topologies – compatibility and interoperability – can be easily adapted to cope with the presence of non-synchronous interactions.

In the case of asynchronous interactions, the semantic model underlying a process algebraic architectural description may have infinitely many states because the additional implicit components behave like unbounded buffers. In order for the modified architectural checks to be effectively applicable in this case, one option is to allow users to limit the size of buffers statically. Another option is to derive sufficient conditions under which the state space is guaranteed to be finite. This will be the subject of future work.

References

1. A. Aldini and M. Bernardo, *"On the Usability of Process Algebra: An Architectural View"*, in Theoretical Computer Science 335:281-329, 2005.
2. R. Allen and D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM Trans. on Software Engineering and Methodology 6:213-249, 1997.
3. M. Bernardo, P. Ciancarini, and L. Donatiello, *"Architecting Families of Software Systems with Process Algebras"*, in ACM Trans. on Software Engineering and Methodology 11:386-426, 2002.
4. C. Canal, E. Pimentel, and J.M. Troya, *"Compatibility and Inheritance in Software Architectures"*, in Science of Computer Programming 41:105-138, 2001.
5. A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, *"Design and Evaluation of a Wide-Area Event Notification Service"*, in ACM Trans. on Computer Systems 19:332-383, 2001.
6. D. Gelernter, *"Generative Communication in Linda"*, in ACM Trans. on Programming Languages and Systems 7:80-112, 1985.
7. P. Inverardi, A.L. Wolf, and D. Yankelevich, *"Static Checking of System Behaviors Using Derived Component Assumptions"*, in ACM Trans. on Software Engineering and Methodology 9:239-272, 2000.
8. J. Magee and J. Kramer, *"Concurrency: State Models & Java Programs"*, Wiley, 1999.
9. R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989.
10. F. Oquendo, *"π-ADL: An Architecture Description Language Based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures"*, in ACM Software Engineering Notes 29(3):1-14, 2004.

Appendix

Operational semantics for PA. Observed that $\llbracket \underline{0} \rrbracket$ is a single-state graph with no transitions, as shown below we have one basic rule for action prefix and several inductive rules for the other operators. Process $a.P$ can execute an action with name a and then behaves as P. Constant B behaves as the process term occurring in its defining equation. $P_1 + P_2$ behaves as either P_1 or P_2 depending on which of them executes an action first (nondeterministic choice). $P_1 \parallel_S P_2$ behaves as P_1 in parallel with P_2 as long as actions are executed whose name does not belong to S, while synchronizations are forced between any action executed by P_1 and any action executed by P_2 that have the same name belonging to S. Finally, P / H behaves as P with all executed actions occurring in H made invisible, whereas $P[\varphi]$ behaves as P with all executed actions relabeled via φ.

$$a.P \xrightarrow{a} P \qquad \frac{B \triangleq P \quad P \xrightarrow{a} P'}{B \xrightarrow{a} P'}$$

$$\frac{P_1 \xrightarrow{a} P'}{P_1 + P_2 \xrightarrow{a} P'} \qquad \frac{P_2 \xrightarrow{a} P'}{P_1 + P_2 \xrightarrow{a} P'}$$

$$\frac{P_1 \xrightarrow{a} P'_1 \quad a \notin S}{P_1 \parallel_S P_2 \xrightarrow{a} P'_1 \parallel_S P_2} \qquad \frac{P_2 \xrightarrow{a} P'_2 \quad a \notin S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1 \parallel_S P'_2}$$

$$\frac{P_1 \xrightarrow{a} P'_1 \quad P_2 \xrightarrow{a} P'_2 \quad a \in S}{P_1 \parallel_S P_2 \xrightarrow{a} P'_1 \parallel_S P'_2}$$

$$\frac{P \xrightarrow{a} P' \quad a \in H}{P / H \xrightarrow{\tau} P' / H} \qquad \frac{P \xrightarrow{a} P' \quad a \notin H}{P / H \xrightarrow{a} P' / H}$$

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

Proof of Thm. 1. Since a star cannot contain cycles, there cannot be attachments between interactions of the AEIs belonging to the border $\mathcal{B}_K$ of the star. Therefore $\llbracket K, \mathcal{B}_K \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K} / H$ is given by:

$$(\llbracket K \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{S_1} \llbracket C_1 \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{S_2} \llbracket C_2 \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{S_3} \dots \parallel_{S_n} \llbracket C_n \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K}) / H$$

where S_j is simply $\mathcal{S}(K, C_j; K, \mathcal{B}_K)$.

From $H_j \cap H_g = \emptyset$ for all $j \neq g$, it follows that the hiding of H can be distributed in such a way that each subset of H is applied as soon as possible. Thus $\llbracket K, \mathcal{B}_K \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K} / H$ is the same as:

$$((\dots((\llbracket K \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{S_1} \llbracket C_1 \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K}) / H_1 \parallel_{S_2} \llbracket C_2 \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K}) / H_2 \parallel_{S_3} \dots) / H_{n-1} \parallel_{S_n} \llbracket C_n \rrbracket^{\mathrm{c}}_{K,\mathcal{B}_K}) / H_n$$

From the compatibility of K with C_1, i.e. $([\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{S_1} [\![C_1]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}) \,/\, H_1 \;\approx_{\mathrm{B}}\; [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H_1$, and the congruence property of $\approx_{\mathrm{B}}$ with respect to static operators, it is possible to replace $([\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{S_1} [\![C_1]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}) \,/\, H_1$ with $[\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H_1$ in $[\![K, \mathcal{B}_K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H$ without changing the semantics of the whole star.

From the compatibility of K with C_2, i.e. $([\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{S_2} [\![C_2]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}) \,/\, H_2 \;\approx_{\mathrm{B}}\; [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H_2$, and the congruence property of $\approx_{\mathrm{B}}$ with respect to static operators, it follows that:

$$([\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{S_2} [\![C_2]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}) \,/\, H_2 \,/\, H_1 \;\approx_{\mathrm{B}}\; [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H_2 \,/\, H_1$$

Since $H_1 \cap H_2 = \emptyset$, the hiding of H_1 has no effect on $[\![C_2]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}$ and hence can be anticipated, thus obtaining:

$$([\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H_1 \parallel_{S_2} [\![C_2]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}) \,/\, H_2 \;\approx_{\mathrm{B}}\; [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H_1 \cup H_2$$

As a consequence, $(([\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \parallel_{S_1} [\![C_1]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}) \,/\, H_1 \parallel_{S_2} [\![C_2]\!]^{\mathrm{c}}_{K,\mathcal{B}_K}) \,/\, H_2$ can be replaced by $[\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H_1 \cup H_2$ in $[\![K, \mathcal{B}_K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H$ without changing the semantics of the whole star.

By reasoning in the same way as we did on C_2 for each of the remanining AEIs in the border, we derive that $[\![K, \mathcal{B}_K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H \;\approx_{\mathrm{B}}\; [\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H$, from which the deadlock freedom of the whole star follows as $[\![K]\!]^{\mathrm{c}}_{K,\mathcal{B}_K} \,/\, H$ is deadlock free by hypothesis and $\approx_{\mathrm{B}}$ preserves deadlock freedom.

(Proceeding by induction on the size of $\mathcal{B}_K$ is hampered by the variability of the set of AEIs with respect to which the interacting semantics are defined.)

Proof of Thm. 2. A straightforward consequence of Def. 2 and the fact that $\approx_{\mathrm{B}}$ preserves deadlock freedom.

Completing the PADL specification of the cruise control system. The speed controller triggers the speed actuator on the basis of the driver commands received through the sensor. It can be in one of the following four states: inactive (when the engine is off), active (when the engine is on), cruising (after pressing the on button in the active state or the resume button in the suspended state), and suspended (after pressing any pedal or button different from on/resume in the cruising state). The speed controller AET is defined as follows:

```
ARCHI_ELEM_TYPE Controller_Type(void)
  BEHAVIOR
    Inactive(void; void) =
      turned_engine_on . Active();
    Active(void; void) =
      choice
      {
        pressed_accelerator . Active(),
        pressed_brake . Active(),
        pressed_on . trigger_record . Cruising(),
        pressed_off . Active(),
        pressed_resume . Active(),
        turned_engine_off . Inactive()
      };
    Cruising(void; void) =
      choice
      {
        pressed_accelerator . trigger_disable . Suspended(),
        pressed_brake . trigger_disable . Suspended(),
        pressed_on . Cruising(),
        pressed_off . trigger_disable . Suspended(),
        pressed_resume . Cruising(),
        turned_engine_off . trigger_disable . Inactive()
      };
    Suspended(void; void) =
      choice
      {
        pressed_accelerator . Suspended(),
        pressed_brake . Suspended(),
        pressed_on . trigger_record . Cruising(),
        pressed_off . Suspended(),
        pressed_resume . trigger_resume . Cruising(),
        turned_engine_off . Inactive()
      }
  INPUT_INTERACTIONS  UNI turned_engine_on; turned_engine_off;
                          pressed_accelerator; pressed_brake;
                          pressed_on; pressed_off; pressed_resume
  OUTPUT_INTERACTIONS UNI trigger_record; trigger_resume;
                          trigger_disable
```

The speed detector periodically communicates the number of wheel revolutions per time unit to the speed actuator. The speed detector AET is defined as follows:

```
ARCHI_ELEM_TYPE Detector_Type(void)
  BEHAVIOR
    Detector_Off(void; void) =
      turned_engine_on . Detector_On();
    Detector_On(void; void) =
      choice
      {
        measure_speed . signal_speed . Detector_On(),
        turned_engine_off . Detector_Off()
      }
  INPUT_INTERACTIONS  UNI turned_engine_on; turned_engine_off
  OUTPUT_INTERACTIONS UNI signal_speed
```

The speed actuator adjusts the throttle on the basis of the triggers received from the controller and of the speed measured by the detector. It can be in one of the following two states: disabled (until the on/resume button is pressed) and enabled (until any pedal or button different from on/resume is pressed). The speed actuator AET is defined as follows:

```
ARCHI_ELEM_TYPE Actuator_Type(void)
  BEHAVIOR
    Disabled(void; void) =
      choice
      {
        signalled_speed . Disabled(),
        triggered_record . record_speed . Enabled(),
        triggered_resume . resume_speed . Enabled()
      };
    Enabled(void; void) =
      choice
      {
        signalled_speed . adjust_throttle . Enabled(),
        triggered_disable . disable_control . Disabled()
      }
  INPUT_INTERACTIONS  UNI triggered_record; triggered_resume;
                          triggered_disable;
                          signalled_speed
  OUTPUT_INTERACTIONS void
```

Finally, the section ARCHI_TOPOLOGY contains the declaration of the four software components together with the attachments among their local interactions, which result in a cyclic topology:

```
ARCHI_ELEM_INSTANCES
  S : Sensor_Type();
  C : Controller_Type();
  D : Detector_Type();
  A : Actuator_Type()
ARCHI_INTERACTIONS
  void
ARCHI_ATTACHMENTS
  FROM S.turn_engine_on    TO C.turned_engine_on;
  FROM S.turn_engine_on    TO D.turned_engine_on;
  FROM S.turn_engine_off   TO C.turned_engine_off;
  FROM S.turn_engine_off   TO D.turned_engine_off;
  FROM S.press_accelerator TO C.pressed_accelerator;
  FROM S.press_brake       TO C.pressed_brake;
  FROM S.press_on          TO C.pressed_on;
  FROM S.press_off         TO C.pressed_off;
  FROM S.press_resume      TO C.pressed_resume;
  FROM C.trigger_record    TO A.triggered_record;
  FROM C.trigger_resume    TO A.triggered_resume;
  FROM C.trigger_disable   TO A.triggered_disable;
  FROM D.signal_speed      TO A.signalled_speed
```