Handling Communications in Process Algebraic Architectural Description Languages: Modeling, Verification, and Implementation

Marco Bernardo Edoardo Bontà Alessandro Aldini

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. If based on formal methods, they can also serve for enabling the early verification of various properties such as component coordination and for guiding the synthesis of code correct by construction. This is the case with process algebraic architectural description languages, which are process calculi enhanced with the main architectural concepts. However, the techniques with which those languages have been equipped are mainly conceived to work with synchronous communications only. The objective of this paper is threefold. 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 nonsynchronous communications in such a way that the usability of the original language is preserved. On the verification side, we show how to modify techniques for analyzing the absence of coordination mismatches like the compatibility check for acyclic topologies and the interoperability check for cyclic topologies in such a way that those checks are valid also for nonsynchronous communications. On the implementation side, we show how to generate multithreaded object-oriented software in the presence of synchronous and nonsynchronous communications in such a way that the properties proved at the architectural level are preserved at the code level.

Key words: software architecture, architectural description languages, process algebra, synchronous and nonsynchronous communications, system modeling and verification, code generation.

1. Introduction

The growing complexity and the increasing size of modern software systems can be managed by adopting notations for formal or semi-formal system modeling (model-driven approach). In this way, design documents with a precise syntax can be produced and shared by all the people contributing to system development. In order to avoid delays and cost increases due to the late discovery of errors in the development process, another task that such notations should carry out is to enable the rigorous and hopefully automated analysis of system properties and to guide the synthesis of code correct by construction. For instance, it is widely recognized that property verification finds its own rightful place in the architectural design phase [38, 12]. The reason is that this phase precedes system implementation and provides support for declarative/behavioral/topological system models that are complete at a high level of abstraction.

Many architectural description languages have been proposed. Some of them – like, e.g., Wright [4, 3], Darwin/FSP [31, 32], LEDA [14], PADL [1], and π-ADL [35] – are based on process algebra [34, 29, 6] due to its support to compositional modeling. It is worth noting that process algebra is compositional, but not component-oriented. Thus, from the point of view of process algebra, its architectural versions are a significant step forward in terms of usability. In fact, they give special prominence to the main architectural concepts – components, connectors, and styles – while hiding the process algebraic technicalities to the software developer.

On the modeling side, this architectural upgrade has three important consequences. First, it permits to describe the behavior of the components separately from the representation of the system topology, thus overcoming the modeling difficulties deriving from the direct use of certain process algebraic operators like, e.g., parallel composition. Second, it highlights the interactions among components and the classification of their communications, thus allowing for static checks establishing system model well-formedness. Third, it fosters the reuse of the specification of single components as well as of complete systems, thus supporting the compositional and hierarchical modeling of entire system families.

On the verification side, process algebraic architectural description languages inherit all the techniques applicable to process algebra, like model checking [18] and equivalence checking [19]. In addition, such languages are equipped with ad-hoc analysis techniques (see, e.g., [4, 30, 15, 1]) mostly based on behavioral equivalences [27], which are useful for detecting coordination mismatches that may arise when assembling together components that are correct if taken in isolation. Moreover, they can generate diagnostic information for pinpointing components responsible for mismatches.

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. The limitation to synchronous communications is not so restrictive for usual properties like deadlock freedom, which should hold when no communication is blocking. In contrast, the validity of other properties related to activity sequencing or message ordering may not be guaranteed in the presence of nonsynchronous communications.

In order to address the usual properties in a more general setting as well as the other properties mentioned above, 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 nonsynchronous 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 input or read probes [25]. A semi-synchronous output port can instead be used to model the interplay between a graphical user interface and an underlying application, as the former must not block 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 [16].

In the extended language, semi-synchronous ports can be easily handled with suitable semantic rules generating exceptions whenever necessary, whereas asynchronous ports require the addition of implicit repository-like components. In any case, the semantic treatment of nonsynchronous communications is completely transparent to PADL users, as they only have to specify suitable synchronicity-related qualifiers in their architectural descriptions. Therefore, the degree of usability of the original language is unaffected.

The second contribution of this paper is to show how to modify techniques for analyzing the absence of coordination mismatches – like the compatibility check for acyclic topologies and the interoperability check for cyclic topologies introduced in [1] – in such a way that those checks can still be applied in the presence of nonsynchronous communications.

This is accomplished by viewing certain activities carried out through semi-synchronous and asynchronous ports as internal activities when performing the above mentioned checks. The reason is that each such activity has a specific outcome and takes place at a specific time instant when considered from the point of view of the individual component executing that activity. However, in the overall architecture the same activity can raise an exception (if the port is semi-synchronous and the other ports are not ready to communicate with it) or can be delayed (if the port is asynchronous and the communication is buffered). Thus, if we do not regard exceptions and all the activities carried out through asynchronous ports as internal activities at verification time, the compatibility or interoperability check may fail even in the absence of a real coordination mismatch.

The third contribution of this paper is to show how to generate multithreaded object-oriented software from process algebraic architectural descriptions including various kinds of communications in such a way that the properties proved at the architectural level are preserved at the code level. This last contribution is related to one of the big issues in the software engineering field, i.e., guaranteeing that the implementation of a software system complies with its architectural description [23]. Indeed, the purpose of automatic code generation should be not only to speed up system implementation, but also to ensure conformance by construction.

In order to bridge the gap between system modeling/verification and system implementation, we propose an approach that automatically synthesizes multithreaded Java programs from PADL descriptions containing an arbitrary combination of synchronous, semi-synchronous, and asynchronous ports. The choice of Java as target language is due to the fact that Java offers a set of mechanisms for the well-structured management of threads and their shared data, which should simplify the code generation task. Moreover, its object-oriented nature – and specifically its encapsulation capability – makes Java an appropriate candidate for coping with the high level of abstraction of process algebraic architectural descriptions during code generation.

The proposed approach is divided into two phases. In the first phase, we develop an architecture-driven technique for thread coordination management. Similar to previous work (see, e.g., [37]), we advocate the provision of a suitable software package that takes care of the details of thread coordination by means of architecture-inspired units in a way that is completely transparent to the software developer. The distinguishing feature of the first phase of our proposal is that also the employment of the package should follow the same architecture-centric spirit. In other words, the use of the package units should be guided by the architectural description of the system to be developed, as this description is a well suited tool for achieving correct thread coordination in the case of concurrent object-oriented programs.

In the second phase, we handle the translation of the process algebraically specified behavior of individual software components into threads. The separation of thread behavior generation from thread coordination management turns out to be particularly appropriate in order to limit human intervention. In fact, while a completely automated and architecture-driven technique can guarantee correct thread coordination, only a partial translation based on stubs is possible for the generation of threads. In addition to the considerations of [32], in which it is shown how a disciplined process algebraic modeling is beneficial at subsequent stages, we also provide a set of guidelines for filling in stubs, which guarantee the preservation at the code level of properties proved on the architectural description of the system under construction.

This paper, which is a full and revised version of [10, 8, 9], is organized as follows. After recalling PADL in Sect. 2, in Sect. 3 we extend its syntax with semi-synchronous and asynchronous ports and we consequently revise its semantics. A running example based on a client-server system is used throughout both sections. In Sect. 4, we modify the architectural compatibility and interoperability checks in order to deal with nonsynchronous communications as well. The modified checks are illustrated on the architectural description of an applet-based simulator for a cruise control system. In Sect. 5, we present the two-phase approach for synthesizing multithreaded Java software from PADL descriptions including synchronous and nonsynchronous communications. The approach is exemplified by means of the same cruise control system simulator as the previous section. Finally, in Sect. 6 we provide some concluding remarks and directions for future work.

2. The Architectural Description Language PADL

PADL is a process algebraic architectural description language. In this section, after recalling some basic notions of process algebra (Sect. 2.1), we present PADL syntax (Sect. 2.2) and semantics (Sect. 2.3) by illustrating them through a client-server running example. For a complete presentation and comparisons with related work, the interested reader is referred to [1].

2.1. Process Algebra

Process calculi [34, 29, 6] provide 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 $\mathit{Name} = \mathit{Name}_{\mathrm{v}} \cup \{\tau\}$ of action names including τ for denoting an invisible action, together with a set $\mathit{Relab} = \{\varphi : \mathit{Name} \rightarrow \mathit{Name} \mid \varphi^{-1}(\tau) = \{\tau\}\}$ of relabeling functions preserving action visibility, we consider a process calculus 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	$(S \subseteq \mathit{Name}_{\mathrm{v}})$
$\mid$	P / H	hiding	$(H \subseteq \mathit{Name}_{\mathrm{v}})$
$\mid$	$P[\varphi]$	relabeling	$(\varphi \in \mathit{Relab})$

Operational semantic rules map every process term P of PA to a state-transition graph $[\![P]\!]$ called labeled transition system. In this graph, 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.

Observed that no rule is necessary for the inactive process $\underline{0}$ – as $[\![\underline{0}]\!]$ must be a single-state graph with no transitions – the operational semantic rules for dynamic operators (action prefix and alternative composition) and process constants are the following:

$$a\,.\,P \xrightarrow{a} P \qquad \frac{B \stackrel{\Delta}{=} P \quad P \xrightarrow{a} P'}{B \xrightarrow{a} P'}$$

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

Process $a\,.\,P$ can execute an action with name a and then behaves as P. Process $P_1 + P_2$ behaves as either P_1 or P_2 depending on which of them executes an action first (nondeterministic choice). Constant B behaves as the process term occurring in its possibly recursive defining equation.

The operational semantic rules for static operators (parallel composition, hiding, and relabeling) are the following:

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

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

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

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

Process $P_1 \parallel_S P_2$ behaves as P_1 in parallel with P_2 as long as actions are executed whose name does not belong to S. In contrast, 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. Process P/H behaves as P with all executed actions occurring in H made invisible. Process $P[\varphi]$ behaves as P with all executed actions relabeled via φ.

Process terms are compared and manipulated by means of behavioral equivalences [27]. 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 [34].

Denoted by $\Longrightarrow$ the extension of $\longrightarrow$ to action sequences, we say that a symmetric relation $\mathcal{R}$ is a weak bisimulation iff for all $(P_1, P_2) \in \mathcal{R}$: *(i)* whenever $P_1 \xrightarrow{a} P_1'$ for $a \in \mathit{Name}_{\mathrm{v}}$, 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 $\approx_{\mathrm{B}}$, defined as the union of all the weak bisimulations, is a congruence with respect to all the operators except for alternative composition and has a modal characterization based on a weak variant of Hennessy-Milner logic.

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 textual description then comprises two sections, as shown below:

```
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
```

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 (behavioral invocations).

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 one establishes whether the interaction is an input or output interaction, whereas the second one 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 like multicasts (qualifier `AND`), or-interactions guiding selective one-to-many communications like those between a server and its clients (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 the last selective many-to-one input was received (keyword `DEP`).

The second section of the PADL description defines the topology of the system family. It is composed of three subsections. 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 a local output interaction of an AEI to a local input interaction of another AEI. Moreover, a local uni-interaction can be attached to only one local interaction, whereas a local and-/or-interaction can be attached to (several) local uni-interactions only.

Besides the textual notation, PADL comes equipped with a graphical notation that is an extension of the flow graph notation [34]. 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). Or-dependences are depicted as dotted edges.

Example 2.1. 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. 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
```

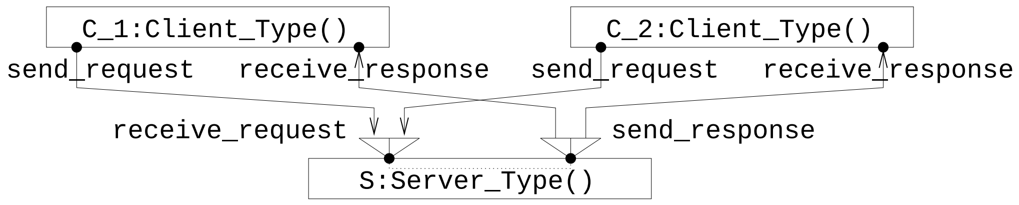
where compute_response is an internal action, while send_response is declared to depend on receive_request in order to make sure that each response is sent back to the client that issued the corresponding request. Since the behavior of the two clients is identical, a single client AET suffices:

```
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. Finally, we declare the topology of the system as follows by using the dot notation for the local interactions:

```
ARCHI_TOPOLOGY
  ARCHI_ELEM_INSTANCES
    S   : Server_Type();
    C_1 : Client_Type();
    C_2 : Client_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
```

The topology is better illustrated by the following enriched flow graph:



where the dotted edge linking the bisected triangles is the or-dependence. ■

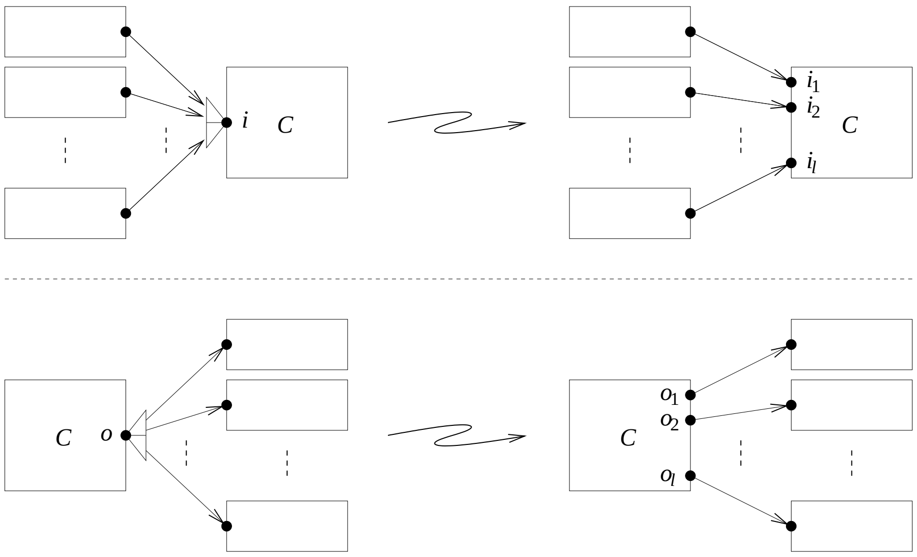
2.3. The Semantics for PADL

The semantics for PADL is given by translation into PA. The first step of the translation focuses on the semantics of each AEI. This is defined as the behavior of the corresponding AET with all the action occurrences being preceded by the name of the AEI and the AET formal data parameters being substituted for by the corresponding AEI actual data parameters.

Let $\mathcal{C}$ be an AET with $m \in \mathbb{N}_{\geq 0}$ formal parameters $fp_1, \dots, fp_m$ and behavior given by a sequence $\mathcal{E}$ of defining equations. Then the semantics of an AEI C of type $\mathcal{C}$ with actual parameters $ap_1, \dots, ap_m$ is defined as follows, where $C\,.\,\_$ introduces the dot notation on actions and $\{\_ \hookrightarrow \_, \dots, \_ \hookrightarrow \_\}$ denotes a syntactical substitution:

$$\boxed{[\![C]\!] \;=\; C.\mathcal{E}\{ap_1 \hookrightarrow fp_1, \dots, ap_m \hookrightarrow fp_m\}}$$

If the AEI contains local or-interactions, each of them – consistent with the fact that it guides a selective one-to-many communication – must be replaced by as many fresh local uni-interactions as there are attachments involving the considered or-interaction. The fresh local uni-interactions are then attached to the local uni-interactions of other AEIs to which the local or-interaction was originally attached, as shown below:



In that case, the semantics of C is defined as follows:

$$\boxed{[\![C]\!] \;=\; \textit{or-rewrite}_{\emptyset}(C.\mathcal{E}\{ap_1 \hookrightarrow fp_1, \dots, ap_m \hookrightarrow fp_m\})}$$

In order to reflect the fact that an or-interaction can result in several alternative communications, function *or-rewrite* inductively rewrites the body of any resulting defining equation by replacing with fresh uni-interactions each occurrence of any local or-interaction whose number *attach-no*(_) of attachments is greater than 1. Or-dependences are dealt with by keeping track of the set *FI* of fresh input uni-interactions currently in force that arise from input or-interactions on which some output or-interaction depends. Here are the three basic clauses of the inductive definition of *or-rewrite*:

- If a is either a local input or-interaction of C on which no local output or-interaction depends, or a local output or-interaction that does not depend on any local input or-interaction, and $\mathit{attach\text{-}no}(C.a) = l \geq 2$:

$$\boxed{\begin{array}{rcl} \mathit{or\text{-}rewrite}_{\mathit{FI}}(a \,.\, P) & = & \texttt{choice}\{a_1 \,.\, \mathit{or\text{-}rewrite}_{\mathit{FI}}(P), \\ & & \qquad \vdots \\ & & \qquad a_l \,.\, \mathit{or\text{-}rewrite}_{\mathit{FI}}(P)\} \end{array}}$$

- If i is a local input or-interaction of C on which a local output or-interaction depends and $\mathit{attach\text{-}no}(C.i) = l \geq 2$:

$$\boxed{\begin{array}{rcl} \mathit{or\text{-}rewrite}_{\mathit{FI}}(i \,.\, P) & = & \texttt{choice}\{i_1 \,.\, \mathit{or\text{-}rewrite}_{\mathit{FI}\,\cup\,\{i_1\}\,-\,\{i_j \mid 1 \leq j \leq l \wedge j \neq 1\}}(P), \\ & & \qquad \vdots \\ & & \qquad i_l \,.\, \mathit{or\text{-}rewrite}_{\mathit{FI}\,\cup\,\{i_l\}\,-\,\{i_j \mid 1 \leq j \leq l \wedge j \neq l\}}(P)\} \end{array}}$$

- If o is a local output or-interaction of C that depends on the local input or-interaction i and $\mathit{attach\text{-}no}(C.i) = \mathit{attach\text{-}no}(C.o) \geq 2$ and $i_j \in \mathit{FI}$:

$$\boxed{\mathit{or\text{-}rewrite}_{\mathit{FI}}(o \,.\, P) \;=\; o_j \,.\, \mathit{or\text{-}rewrite}_{\mathit{FI}}(P)}$$

Example 2.2. Consider the client-server system described in Ex. 2.1. Then ⟦C_1⟧ and ⟦C_2⟧ coincide with the defining equation for `Client`, where action names are preceded by `C_1` and `C_2`, respectively. In contrast, ⟦S⟧ is given by the following defining equation obtained from the one for `Server` after rewriting the occurring or-interactions:

```
RewSer(void; void) =
 choice
 {
  S.receive_request_1 . S.compute_response . S.send_response_1 . RewSer(),
  S.receive_request_2 . S.compute_response . S.send_response_2 . RewSer()
 }
```
■

The second step of the translation defines the meaning of any set of AEIs $\{C_1, \dots, C_n\}$ and hence of an entire architectural description. Fixed an AEI C_j in the set, let $\mathcal{LI}_{C_j}$ be the set of local interactions of C_j and $\mathcal{LI}_{C_j;C_1,\dots,C_n} \subseteq \mathcal{LI}_{C_j}$ be the set of local interactions of C_j attached to $\{C_1, \dots, C_n\}$.

In order to make the process terms representing the semantics of these AEIs communicate in the presence of attached interactions having different names – in PA only actions with the same name can synchronize – we need a set $\mathcal{S}(C_1, \dots, C_n)$ of fresh action names, one for each pair of attached local uni-interactions in $\{C_1, \dots, C_n\}$ and for each set of local uni-interactions attached to the same local and-interaction in $\{C_1, \dots, C_n\}$. In order to ensure the uniqueness of the elements of $\mathcal{S}(C_1, \dots, C_n)$, each of them is built by concatenating through symbol # all the original names in the corresponding maximal set of attached local interactions. Then, we need suitable injective relabeling functions $\varphi_{C_j;C_1,\dots,C_n}$ mapping each set $\mathcal{LI}_{C_j;C_1,\dots,C_n}$ to $\mathcal{S}(C_1, \dots, C_n)$ in such a way that:

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

if and only if $C_j.a_1$ and $C_g.a_2$ are attached to each other or to the same and-interaction.

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

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

In general, the interacting semantics of $\{C'_1, \dots, C'_{n'}\} \subseteq \{C_1, \dots, C_n\}$ with respect to $\{C_1, \dots, C_n\}$ is defined as follows (assuming parallel composition to be left associative):

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

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 2.3. Consider again the client-server system of Ex. 2.1. Then the semantics of the entire description is given by the following process term:

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

where $[\![\texttt{S}]\!]$, $[\![\texttt{C\_1}]\!]$, and $[\![\texttt{C\_2}]\!]$ have been shown in Ex. 2.2. ■

3. Semi-Synchronous and Asynchronous Interactions

The interactions occurring in a PADL description can be involved only in synchronous communications, hence input and output interactions represent 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 – in which we use the same client-server running example as the previous section – we first enrich the textual and graphical notations in order to express nonsynchronous interactions (Sect. 3.1). Then, we discuss the treatment of semi-synchronous interactions through additional semantic rules (Sect. 3.2) and of asynchronous interactions through additional implicit AEIs (Sect. 3.3). Finally, we revise the semantics accordingly (Sect. 3.4). The nine resulting forms of communication are summarized by Fig. 1, with the first one being the only one originally available in PADL.

3.1. Enriching PADL Textual and Graphical Notations

In the textual notation of PADL, we introduce a third qualifier for each interaction, to be used in the definition of the AETs. Such a qualifier establishes whether the interaction is synchronous (keyword `SYNC`), semi-synchronous (keyword `SSYNC`), or asynchronous (keyword `ASYNC`).

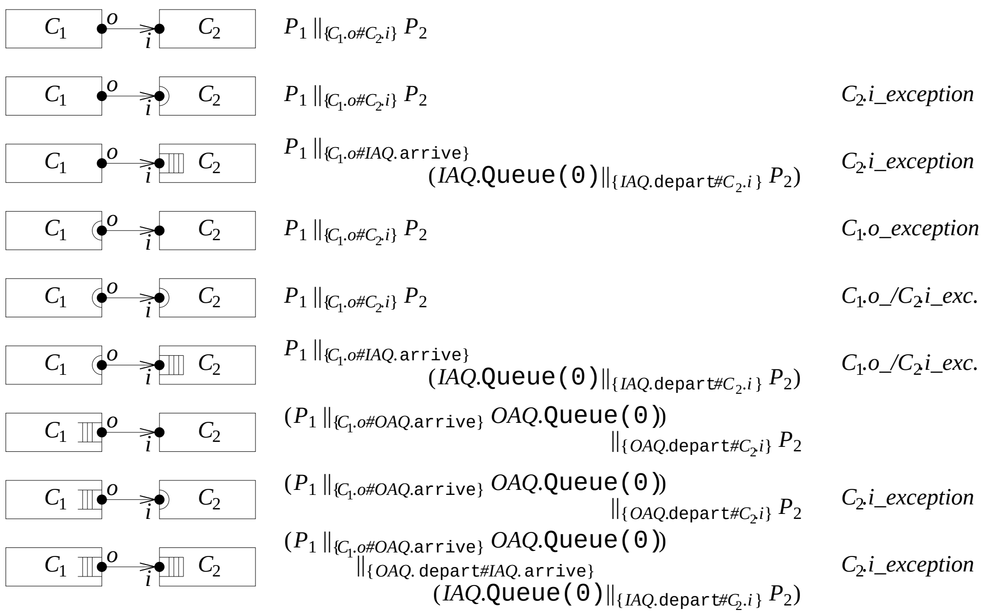

C_1
C_2
o
i
$P_1 \|_{\{C_1.o\#C_2.i\}} P_2$
$P_1 \|_{\{C_1.o\#C_2.i\}} P_2$
$C_2.i\_exception$
$P_1 \|_{\{C_1.o\#IAQ.\texttt{arrive}\}} (IAQ.\texttt{Queue(0)} \|_{\{IAQ.\texttt{depart}\#C_2.i\}} P_2)$
$C_2.i\_exception$
$P_1 \|_{\{C_1.o\#C_2.i\}} P_2$
$C_1.o\_exception$
$P_1 \|_{\{C_1.o\#C_2.i\}} P_2$
$C_1.o\_/C_2.i\_exc.$
$P_1 \|_{\{C_1.o\#IAQ.\texttt{arrive}\}} (IAQ.\texttt{Queue(0)} \|_{\{IAQ.\texttt{depart}\#C_2.i\}} P_2)$
$C_1.o\_/C_2.i\_exc.$
$(P_1 \|_{\{C_1.o\#OAQ.\texttt{arrive}\}} OAQ.\texttt{Queue(0)}) \|_{\{OAQ.\texttt{depart}\#C_2.i\}} P_2$
$(P_1 \|_{\{C_1.o\#OAQ.\texttt{arrive}\}} OAQ.\texttt{Queue(0)}) \|_{\{OAQ.\texttt{depart}\#C_2.i\}} P_2$
$C_2.i\_exception$
$(P_1 \|_{\{C_1.o\#OAQ.\texttt{arrive}\}} OAQ.\texttt{Queue(0)}) \|_{\{OAQ.\texttt{depart}\#IAQ.\texttt{arrive}\}} (IAQ.\texttt{Queue(0)} \|_{\{IAQ.\texttt{depart}\#C_2.i\}} P_2)$
$C_2.i\_exception$

Figure 1: Forms of communications available in the extension of PADL

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 nonsynchronous 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 $s.\texttt{success}$ is associated with each semi-synchronous interaction s. This implicitly declared variable is automatically set at each execution of s and is made available to PADL users in order to catch exceptions. In this way, it is easy to model different behaviors to be undertaken depending on the outcome of the execution of s.

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 3.1. Consider once more the client-server system of Ex. 2.1. 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 achieved by transforming `send_request` into a semi-synchronous interaction and by redefining the behavior of `Client_Type` as follows:

```
ARCHI_ELEM_TYPE Client_Type(void)
  BEHAVIOR
    Client_Internal(void; void) =
      process . Client_Interacting();
    Client_Interacting(void; void) =
      send_request .
        choice
        {
          cond(send_request.success = true) ->
                      receive_response . Client_Internal(),
          cond(send_request.success = false) ->
                      keep_processing . Client_Interacting()
        }
  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 output interaction `send_response` within the definition of `Server_Type`. ■

3.2. Semantics of Semi-Synchronous Interactions: Additional Rules

A local semi-synchronous interaction s executed by an AEI C gives rise to a transition labeled with $C.s$ within $[\![C]\!]$, and hence to the setting of the related `success` variable to true. However, in an interacting context, this transition has to be relabeled with $C.s\_exception$ if s cannot immediately participate in a communication. This is accomplished by means of additional semantic rules encoding a context-sensitive variant of the relabeling operator.

Suppose that the local output interaction o of an AEI C_1 is attached to the local input interaction i of an AEI C_2, where $C_1.o\#C_2.i$ is their fresh name. Let 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:

$$\boxed{\dfrac{P_1 \not\xrightarrow{C_1.o\#C_2.i} P_1' \qquad P_2 \xrightarrow{C_1.o\#C_2.i} P_2'}{P_1 \,\|_S\, P_2 \xrightarrow{C_2.i\_exception} P_1 \,\|_S\, P_2' \qquad C_2.i.\texttt{success} = \texttt{false}}}$$

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:

$$\boxed{\dfrac{P_1 \xrightarrow{C_1.o\#C_2.i} P_1' \qquad P_2 \not\xrightarrow{C_1.o\#C_2.i} P_2'}{P_1 \,\|_S\, P_2 \xrightarrow{C_1.o\_exception} P_1' \,\|_S\, P_2 \qquad C_1.o.\texttt{success} = \texttt{false}}}$$

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 additional semantic rules together.

3.3. Semantics of Asynchronous Interactions: Additional 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 local asynchronous uni-/and-interaction of an AEI C we have to introduce additional implicit AEIs that behave like unbounded buffers, thus realizing the third, sixth, seventh, eighth, and ninth form of communication depicted in Fig. 1. As shown in Fig. 2, in the case of a local asynchronous and-interaction, it is necessary to introduce as many additional implicit AEIs as there are attachments involving the and-interaction.[1]

Each additional implicit input asynchronous queue (IAQ) and output asynchronous queue (OAQ) is of the following type, where `arrive` is an always-enabled input synchronous uni-interaction while `depart` is an output synchronous uni-interaction enabled only if the buffer is not empty:

[1]In [10], a single additional implicit AEI was introduced even in the case of a local asynchronous and-interaction, thus determining an unnecessary synchronization among the AEIs attached to the and-interaction.

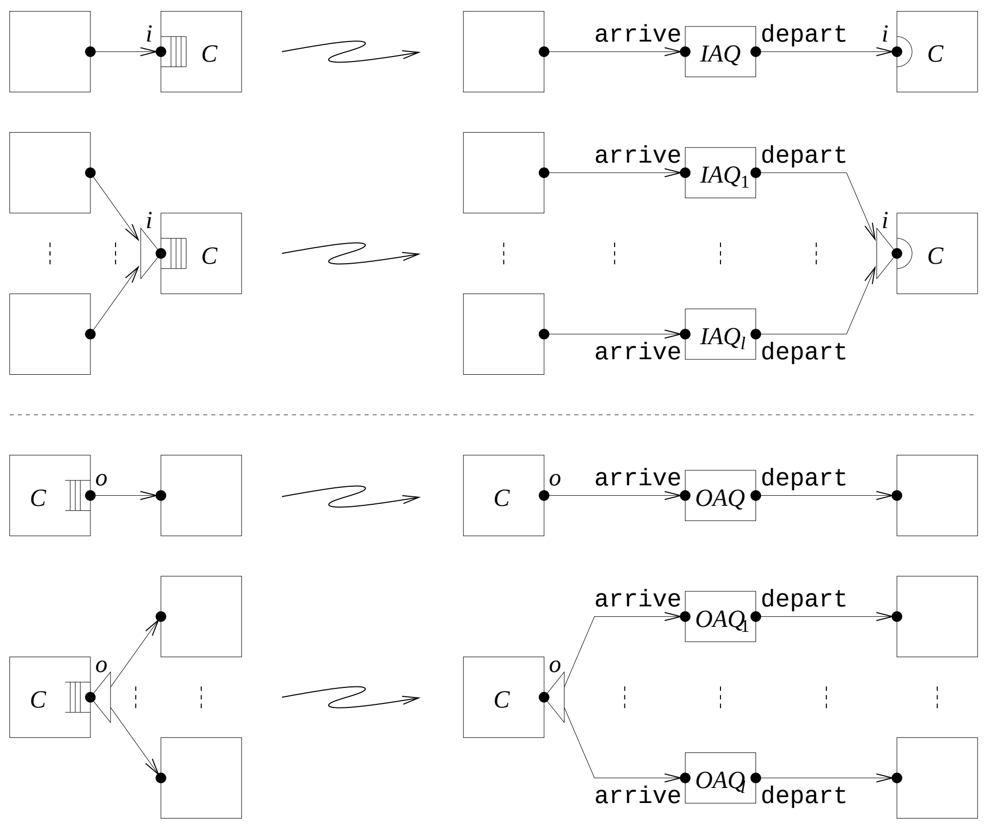


Figure 2: Topological management of local asynchronous uni-/and-interactions

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

In the case of a local input asynchronous uni-/and-interaction i of C, each local output uni-interaction originally attached to i is implicitly re-attached to the `arrive` interaction of one of the additional implicit IAQs. By contrast, the `depart` interaction of each additional implicit IAQ is implicitly attached to i, which is implicitly converted into a semi-synchronous interaction. Note that i becomes semi-synchronous because the communications between the `depart` interactions and i must not block C whenever some buffers are empty.

In the case of a local output asynchronous uni-/and-interaction o of C, this interaction is implicitly converted into a synchronous interaction and reattached to each `arrive` interaction of the additional implicit OAQs. Note that o is never blocked because all `arrive` interactions are always enabled. By contrast, the `depart` interaction of each additional implicit OAQ is attached to one of the input interactions originally attached to o.

3.4. Revising PADL Semantics

Due to the way nonsynchronous interactions have been managed, 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 local asynchronous interactions.

Definition 3.2. Let $\mathcal{C}$ be an AET, $\mathit{fp}_1, \dots, \mathit{fp}_m$ be its $m \in \mathbb{N}_{\geq 0}$ formal parameters, and $\mathcal{E}$ be a sequence of defining equations giving its behavior. Let C be an AEI of type $\mathcal{C}$ with actual parameters $\mathit{ap}_1, \dots, \mathit{ap}_m$ consistent by order and type with the formal parameters. Suppose that C has:

- $h \in \mathbb{N}_{\geq 0}$ local input asynchronous uni-interactions $i_1, \dots, i_h$ handled through the related additional implicit AEIs $IAQ_1, \dots, IAQ_h$;
- $h' \in \mathbb{N}_{\geq 0}$ local input asynchronous and-interactions $i'_1, \dots, i'_{h'}$, where each i'_j is handled through the $\mathit{attach\text{-}no}(C.i'_j) = \mathit{il}_j$ related additional implicit AEIs $IAQ_{j,1}, \dots, IAQ_{j,\mathit{il}_j}$;
- $k \in \mathbb{N}_{\geq 0}$ local output asynchronous uni-interactions $o_1, \dots, o_k$ handled through the related additional implicit AEIs $OAQ_1, \dots, OAQ_k$;
- $k' \in \mathbb{N}_{\geq 0}$ local output asynchronous and-interactions $o'_1, \dots, o'_{k'}$, where each o'_j is handled through the $\mathit{attach\text{-}no}(C.o'_j) = \mathit{ol}_j$ related additional implicit AEIs $OAQ_{j,1}, \dots, OAQ_{j,\mathit{ol}_j}$.

Then the semantics of C is the result of a cascade of function applications:

$$\boxed{[\![C]\!] \; = \; \mathit{o\text{-}and}^{k'}_{\mathit{ol}_{k'}}(\dots \mathit{o\text{-}and}^{1}_{\mathit{ol}_1}(\mathit{o\text{-}uni}_k(\mathit{i\text{-}and}^{h'}_{\mathit{il}_{h'}}(\dots \mathit{i\text{-}and}^{1}_{\mathit{il}_1}(\mathit{i\text{-}uni}_h(C))\dots)))\dots)}$$

where, denoted by $f(C)$ the current result, we define:

$$
\begin{array}{|rcll|}
\hline
\textit{i-uni}_0(C) & = & \textit{or-rewrite}_\emptyset(C.\mathcal{E}\{ap_1 \hookrightarrow fp_1, \dots, ap_m \hookrightarrow fp_m\})\,[\varphi_{C,\text{async}}] & \\
\textit{i-uni}_j(C) & = & IAQ_j.\texttt{Queue(0)}\,[\varphi_{C,\text{async}}] & 1 \leq j \leq h \\
 & & \quad \|_{\{IAQ_j.\texttt{depart}\#C.i_j\}}\,(\textit{i-uni}_{j-1}(C)) & \\
\hline
\textit{i-and}^j_1(f(C)) & = & IAQ_{j,1}.\texttt{Queue(0)}\,[\varphi_{C,\text{async}}] & 1 \leq j \leq h' \\
 & & \quad \|_{\{IAQ_{j,1}.\texttt{depart}\#\dots\#IAQ_{j,il_j}.\texttt{depart}\#C.i'_j\}}\,(f(C)) & \\
\textit{i-and}^j_{j'}(f(C)) & = & IAQ_{j,j'}.\texttt{Queue(0)}\,[\varphi_{C,\text{async}}] & 2 \leq j' \leq il_j \\
 & & \quad \|_{\{IAQ_{j,1}.\texttt{depart}\#\dots\#IAQ_{j,il_j}.\texttt{depart}\#C.i'_j\}}\,(\textit{i-and}^j_{j'-1}(f(C))) & \\
\hline
\textit{o-uni}_0(f(C)) & = & f(C) & \\
\textit{o-uni}_j(f(C)) & = & (\textit{o-uni}_{j-1}(f(C)))\,\|_{\{C.o_j\#OAQ_j.\texttt{arrive}\}} & \\
 & & \quad OAQ_j.\texttt{Queue(0)}\,[\varphi_{C,\text{async}}] & 1 \leq j \leq k \\
\hline
\textit{o-and}^j_1(f(C)) & = & (f(C))\,\|_{\{C.o'_j\#OAQ_{j,1}.\texttt{arrive}\#\dots\#OAQ_{j,ol_j}.\texttt{arrive}\}} & \\
 & & \quad OAQ_{j,1}.\texttt{Queue(0)}\,[\varphi_{C,\text{async}}] & 1 \leq j \leq k' \\
\textit{o-and}^j_{j'}(f(C)) & = & (\textit{o-and}^j_{j'-1}(f(C)))\,\|_{\{C.o'_j\#OAQ_{j,1}.\texttt{arrive}\#\dots\#OAQ_{j,ol_j}.\texttt{arrive}\}} & \\
 & & \quad OAQ_{j,j'}.\texttt{Queue(0)}\,[\varphi_{C,\text{async}}] & 2 \leq j' \leq ol_j \\
\hline
\end{array}
$$

with relabeling function $\varphi_{C,\text{async}}$ transforming the originally asynchronous local interactions of C and the local interactions of the additional implicit AEIs attached to them into the respective fresh names occurring in the synchronization sets above. ■

Example 3.3. Consider the variant of the client-server system described in Ex. 3.1. With respect to Ex. 2.2, $[\![S]\!]$ changes as follows due to the presence of its two local output asynchronous uni-interactions:

```
RewSer[S.send_response_1 ↦ S.send_response_1#OAQ_1.arrive,
       S.send_response_2 ↦ S.send_response_2#OAQ_2.arrive]
  ||{S.send_response_1#OAQ_1.arrive}
    OAQ_1.Queue(0)[OAQ_1.arrive ↦ S.send_response_1#OAQ_1.arrive]
      ||{S.send_response_2#OAQ_2.arrive}
        OAQ_2.Queue(0)[OAQ_2.arrive ↦ S.send_response_2#OAQ_2.arrive]
```

As a consequence, with respect to Ex. 2.3, the semantics of the whole description changes as follows, where the roles of `S.send_response_1` and `S.send_response_2` are now played by `AOQ_1.depart` and `AOQ_2.depart`, respectively:

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

4. Modifying Architectural Checks

The objective of the architectural checks for PADL developed in [1] is to infer certain architectural properties like correct component coordination from the properties of the individual AEIs. The idea is to verify the absence of coordination mismatches resulting in property violations through a topological reduction process based on equivalence checking. Given an architectural description, the starting point is constituted by an abstract variant of its enriched flow graph, where vertices correspond to AEIs and two vertices are linked by an edge if and only if attachments have been declared among their interactions. From a topological viewpoint, the resulting graph is a combination of possibly intersecting stars (see Sect. 4.2) and cycles (see Sect. 4.3), which are thus viewed as basic topological formats.

The strategy then consists of applying specific checks locally to all stars and cycles occurring in the abstract graph. Each check verifies whether the star/cycle contains an AEI behaviorally equivalent to the whole star/cycle, in which case the star/cycle can be replaced by that AEI. The process successfully terminates when the whole graph has been reduced to a single behaviorally equivalent AEI, as at that point it is sufficient to verify whether that AEI satisfies the given property or not. In case of failure, the mentioned checks provide diagnostic information useful to pinpoint components responsible for possible property violations within a single star/cycle.

In order to be applicable, the strategy requires the existence of a behavioral equivalence that possesses the following characteristics. Firstly, the equivalence must preserve the property of interest – i.e., it cannot relate two models such that one of them enjoys the property whereas the other one does not – which is fundamental for enabling the topological reduction process. Secondly, it must be a congruence with respect to static operators, thus allowing the topological reduction process to be applied to single portions of the topology of an architectural description – which is more efficient than considering the entire topology at once – without affecting the possible validity of the property. Thirdly, it must be able to abstract from internal actions, as an architectural property is typically expressed in terms of the possibility/necessity of executing local interactions in a certain order. Fourthly, it must have a modal logic characterization, which is necessary for producing diagnostic information in case of failure of the topological reduction process.

In this section, after introducing some further notation (Sect. 4.1), we show how to modify the compatibility check for stars (Sect. 4.2) and the interoperability check for cycles (Sect. 4.3) in such a way that both checks can still be applied in the presence of nonsynchronous interactions. Then, the two revised checks are exemplified on the architectural description of an applet-based simulator for a cruise control system (Sect. 4.4). Although these checks are conceived for an entire class of properties characterizable through behavioral equivalences that meet the four constraints mentioned above, 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 $\approx_{\rm B}$ introduced in Sect. 2.1.

4.1. Revising Closed Interacting Semantics

Before applying the architectural checks to a star/cycle given by the set of AEIs $\{C_1, \dots, C_n\}$, for each AEI C_j in the set we have to hide all of its internal actions and architectural interactions as well as all of its local interactions that are not attached to $\{C_1, \dots, C_n\}$. The reason is that these actions cannot result in mismatches within the star/cycle, but may hamper the topological reduction process if left visible. Following the terminology of [1], we thus have to consider closed variants of the interacting semantics of the AEIs in the set, in which the mentioned actions are made unobservable.

In the framework of PADL enriched with nonsynchronous interactions, for each AEI C_j in the set we also have to hide all of its additional implicit AEIs that are not attached to $\{C_1, \dots, C_n\}$, as those additional implicit AEIs are necessary only in the presence of the AEIs not in $\{C_1, \dots, C_n\}$ to which they are attached. Therefore, the only actions that remain observable are those in $\mathcal{LI}_{C_j;C_1,\dots,C_n}$ and those in $\mathcal{OALI}_{C_j}$. The latter set contains the originally asynchronous local interactions of C_j together with the local interactions of the related additional implicit AEIs to which they have been re-attached, including the exceptions that may be raised by the local input semi-synchronous interactions in the set corresponding to local input asynchronous interactions. We point out that $\mathcal{OALI}_{C_j}$ is disjoint from $\mathcal{LI}_{C_j}$, as it essentially comprises the action names forming the composite names occurring in the synchronization sets of the semantics of an AEI in isolation provided in Def. 3.2.

In order to set the visibility of each action of C_j according to the needs of the topological reduction process, we introduce a partially closed variant of the interacting semantics of an AEI defined at the end of Sect. 2.3, in which we hide all the actions not in $\mathcal{LI}_{C_j;C_1,\dots,C_n} \cup \mathcal{OALI}_{C_j}$. Since in many cases we also have to hide all the actions in $\mathcal{OALI}_{C_j}$, we introduce a totally closed variant too. Thus, unlike [1], we have two closed variants of the interacting semantics of an AEI. Both variants are parameterized with respect to a set of AEIs $\{C''_1, \dots, C''_{n''}\}$, $n'' \in \mathbb{N}$, determining the additional implicit AEIs to be included.

Definition 4.1. Let $\mathcal{C}$ be an AET, $fp_1, \dots, fp_m$ be its $m \in \mathbb{N}_{\geq 0}$ formal parameters, and $\mathcal{E}$ be a sequence of defining equations giving its behavior. Let $C_j \in \{C_1, \dots, C_n\}$ be an AEI of type $\mathcal{C}$ with actual parameters $ap_1, \dots, ap_m$ consistent by order and type with the formal parameters. The interacting semantics of C_j with respect to $\{C_1, \dots, C_n\}$ without buffers for its originally asynchronous local interactions is defined as follows:

$$\boxed{\begin{array}{rcl} [\![C_j]\!]^{\mathrm{wob}}_{C_1,\dots,C_n} & = & \mathit{or\text{-}rewrite}_{\emptyset}(C_j.\mathcal{E}\{ap_1 \hookrightarrow fp_1, \dots, ap_m \hookrightarrow fp_m\}) \\ & & [\varphi_{C_j,\mathrm{async}}]\,[\varphi_{C_j;C_1,\dots,C_n}] \end{array}}$$

We denote by $[\![C_j]\!]^{\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n}$ the variant of $[\![C_j]\!]^{\mathrm{wob}}_{C_1,\dots,C_n}$ including the buffers for the originally asynchronous local interactions of C_j attached to $\{C''_1, \dots, C''_{n''}\}$. ■

Definition 4.2. Let $C_j \in \{C_1, \dots, C_n\}$. The partially closed interacting semantics of C_j with respect to $\{C_1, \dots, C_n\}$ including its buffers attached to $\{C''_1, \dots, C''_{n''}\}$ is defined as follows:

$$\boxed{[\![C_j]\!]^{\mathrm{pc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n} \;=\; [\![C_j]\!]^{\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n} \,/\, (\mathit{Name} - \mathcal{V}_{C_j;C_1,\dots,C_n})}$$

with $\mathcal{V}_{C_j;C_1,\dots,C_n} = \varphi_{C_j;C_1,\dots,C_n}(\mathcal{LI}_{C_j;C_1,\dots,C_n}) \cup \varphi_{C_j,\mathrm{async}}(\mathcal{OALI}_{C_j})$ and we write $[\![C_j]\!]^{\mathrm{pc;wob}}_{C_1,\dots,C_n}$ if $n'' = 0$. ■

Definition 4.3. Let $\{C'_1, \dots, C'_{n'}\} \subseteq \{C_1, \dots, C_n\}$. The partially closed interacting semantics of $\{C'_1, \dots, C'_{n'}\}$ with respect to $\{C_1, \dots, C_n\}$ including their buffers attached to $\{C''_1, \dots, C''_{n''}\}$ is defined as follows:

$$\boxed{\begin{array}{l} \llbracket C'_1, \dots, C'_{n'} \rrbracket^{\mathrm{pc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n} = \\ \qquad\qquad\qquad \llbracket C'_1 \rrbracket^{\mathrm{pc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n} \, \|_{\mathcal{S}(C'_1,C'_2;C_1,\dots,C_n)} \\ \qquad\qquad\qquad\quad \llbracket C'_2 \rrbracket^{\mathrm{pc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n} \, \|_{\mathcal{S}(C'_1,C'_3;C_1,\dots,C_n) \cup \mathcal{S}(C'_2,C'_3;C_1,\dots,C_n)} \cdots \\ \qquad\qquad\qquad\qquad \cdots \, \|_{\bigcup\limits_{i=1}^{n'-1} \mathcal{S}(C'_i,C'_{n'};C_1,\dots,C_n)} \, \llbracket C'_{n'} \rrbracket^{\mathrm{pc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n} \end{array}}$$

where the synchronization sets are built as at the end of Sect. 2.3. ■

Definition 4.4. Let $C_j \in \{C_1, \dots, C_n\}$. The totally closed interacting semantics of C_j with respect to $\{C_1, \dots, C_n\}$ including its buffers attached to $\{C''_1, \dots, C''_{n''}\}$ is defined as follows:

$$\boxed{\llbracket C_j \rrbracket^{\mathrm{tc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n} \; = \; \llbracket C_j \rrbracket^{\mathrm{pc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n} \, / \, \varphi_{C_j,\mathrm{async}}(\mathcal{OALI}_{C_j})}$$

and we write $\llbracket C_j \rrbracket^{\mathrm{tc;wob}}_{C_1,\dots,C_n}$ if $n'' = 0$. ■

Definition 4.5. Let $\{C'_1, \dots, C'_{n'}\} \subseteq \{C_1, \dots, C_n\}$. The totally closed interacting semantics of $\{C'_1, \dots, C'_{n'}\}$ with respect to $\{C_1, \dots, C_n\}$ including their buffers attached to $\{C''_1, \dots, C''_{n''}\}$ is defined as follows:

$$\boxed{\begin{array}{l} \llbracket C'_1, \dots, C'_{n'} \rrbracket^{\mathrm{tc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n} = \\ \qquad\qquad\qquad \llbracket C'_1 \rrbracket^{\mathrm{tc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n} \, \|_{\mathcal{S}(C'_1,C'_2;C_1,\dots,C_n)} \\ \qquad\qquad\qquad\quad \llbracket C'_2 \rrbracket^{\mathrm{tc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n} \, \|_{\mathcal{S}(C'_1,C'_3;C_1,\dots,C_n) \cup \mathcal{S}(C'_2,C'_3;C_1,\dots,C_n)} \cdots \\ \qquad\qquad\qquad\qquad \cdots \, \|_{\bigcup\limits_{i=1}^{n'-1} \mathcal{S}(C'_i,C'_{n'};C_1,\dots,C_n)} \, \llbracket C'_{n'} \rrbracket^{\mathrm{tc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n} \end{array}}$$

where the synchronization sets are built as at the end of Sect. 2.3. The variant totally closed up to $\{C'''_1, \dots, C'''_{n'''}\} \subset \{C'_1, \dots, C'_{n'}\}$, i.e., in which $\llbracket C'''_j \rrbracket^{\mathrm{pc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n}$ is considered in place of $\llbracket C'''_j \rrbracket^{\mathrm{tc};\#C''_1,\dots,C''_{n''}}_{C_1,\dots,C_n}$, is denoted by $\llbracket C'_1, \dots, C'_{n'} \rrbracket^{\mathrm{tc};\#C''_1,\dots,C''_{n''};C'''_1,\dots,C'''_{n'''}}_{C_1,\dots,C_n}$. ■

4.2. Adapting Architectural Compatibility for Stars

A star is a portion of the abstract enriched flow graph of an architectural description, which is not part of a cyclic subgraph. It is formed by a central AEI K and a border $\mathcal{B}_K = \{C_1, \ldots, C_n\}$ including all the AEIs attached to K. As explained in [1], the validity of an architectural property over a star can be investigated by analyzing the interplay between the central AEI K and each of the AEIs in the border, as there cannot be attachments among AEIs in the border. In order to achieve a correct coordination between K and $C_j \in \mathcal{B}_K$, the actual observable behavior of C_j should coincide with the observable behavior expected by K. In other words, the observable behavior of K should not be altered by the insertion of C_j into the border of the star.

In order to cope with the presence of nonsynchronous interactions, we modify the architectural compatibility check for stars as follows.

Definition 4.6. Given an architectural description $\mathcal{A}$, let K be the central AEI of a star of $\mathcal{A}$, $\mathcal{B}_K = \{C_1, \ldots, C_n\}$ be the border of the star, C_j be an AEI in $\mathcal{B}_K$, H_{K,C_j} be the set of interactions of additional implicit AEIs of K that are attached to interactions of C_j, and E_{K,C_j} be the set of exceptions that may be raised by semi-synchronous interactions involved in attachments between K and C_j. We say that K is compatible with C_j iff:

$$([\![K]\!]_{\mathcal{A}}^{\mathrm{pc};\#C_j} \, \|_{\mathcal{S}(K,C_j;\mathcal{A})} \, [\![C_j]\!]_{K,\mathcal{B}_K}^{\mathrm{tc};\#K}) \, / \, (H_{K,C_j} \cup E_{K,C_j}) \;\approx_{\mathrm{B}}\; [\![K]\!]_{\mathcal{A}}^{\mathrm{pc;wob}} \qquad \blacksquare$$

All possible originally asynchronous local interactions of C_j and all of its interactions possibly attached to AEIs outside the star have been hidden by taking the totally closed interacting semantics of C_j with respect to the AEIs inside the star. We also observe that $H_{K,C_j} \cup E_{K,C_j} = \emptyset$ whenever there are no local nonsynchronous interactions involved in attachments inside the star, in which case all partially closed interacting semantics between $[\![K]\!]_{\mathcal{A}}^{\mathrm{pc};\#\mathcal{A}}$ and $[\![K]\!]_{\mathcal{A}}^{\mathrm{pc;wob}}$ coincide with $[\![K]\!]_{\mathcal{A}}^{\mathrm{tc;wob}}$.

We now extend the compatibility theorem of [1] to nonsynchronous interactions. This provides a sufficient condition for reducing the deadlock verification of the entire star to the deadlock verification of its central AEI.

Theorem 4.7. Let $\mathcal{A}$, K, $\mathcal{B}_K$, C_j, H_{K,C_j}, and E_{K,C_j} be the same as Def. 4.6. Whenever K is compatible with every $C_j \in \mathcal{B}_K$, then:

$$[\![K, \mathcal{B}_K]\!]_{K,\mathcal{B}_K}^{\mathrm{tc};\#K,\mathcal{B}_K;K} \, / \, \bigcup_{j=1}^{n} (H_{K,C_j} \cup E_{K,C_j}) \;\approx_{\mathrm{B}}\; [\![K]\!]_{\mathcal{A}}^{\mathrm{pc;wob}}$$

hence $[\![K, \mathcal{B}_K]\!]_{K,\mathcal{B}_K}^{\mathrm{tc};\#K,\mathcal{B}_K;K} \, / \, \bigcup_{j=1}^{n} (H_{K,C_j} \cup E_{K,C_j})$ is deadlock free iff so is $[\![K]\!]_{\mathcal{A}}^{\mathrm{pc;wob}}$.

Proof Since there cannot be attachments between interactions of the AEIs of $\mathcal{B}_K$, it turns out that $[\![K, \mathcal{B}_K]\!]^{\mathrm{tc};\#K,\mathcal{B}_K;K}_{K,\mathcal{B}_K} \;/\; \bigcup\limits_{j=1}^{n}(H_{K,C_j} \cup E_{K,C_j})$ is given by:

$$([\![K]\!]^{\mathrm{pc};\#\mathcal{B}_K}_{\mathcal{A}} \parallel_{\mathcal{S}(K,C_1;\mathcal{A})} [\![C_1]\!]^{\mathrm{tc};\#K}_{K,\mathcal{B}_K} \parallel_{\mathcal{S}(K,C_2;\mathcal{A})} [\![C_2]\!]^{\mathrm{tc};\#K}_{K,\mathcal{B}_K}$$
$$\parallel_{\mathcal{S}(K,C_3;\mathcal{A})} \dots \parallel_{\mathcal{S}(K,C_n;\mathcal{A})} [\![C_n]\!]^{\mathrm{tc};\#K}_{K,\mathcal{B}_K}) / \bigcup_{j=1}^{n}(H_{K,C_j} \cup E_{K,C_j})$$

Since each local asynchronous and-interaction is dealt with by introducing as many additional implicit AEIs as there are attachments involving the and-interaction, $H_{K,C_j} \cap H_{K,C_g} = \emptyset$ for all $j \neq g$. Hence, every hiding set H_{K,C_j} can be distributed in such a way that it is applied as soon as possible. Similarly, every hiding set E_{K,C_j} can be anticipated too, because E_{K,C_j} is independent from E_{K,C_g} for all $j \neq g$ due to the fact that any exception is raised locally at a single AEI. As a consequence, $[\![K, \mathcal{B}_K]\!]^{\mathrm{tc};\#K,\mathcal{B}_K;K}_{K,\mathcal{B}_K} \;/\; \bigcup\limits_{j=1}^{n}(H_{K,C_j} \cup E_{K,C_j})$ can be rewritten in the following way:

$$(\dots(([\![K]\!]^{\mathrm{pc};\#\mathcal{B}_K}_{\mathcal{A}} \parallel_{\mathcal{S}(K,C_1;\mathcal{A})} [\![C_1]\!]^{\mathrm{tc};\#K}_{K,\mathcal{B}_K}) / (H_{K,C_1} \cup E_{K,C_1})$$
$$\parallel_{\mathcal{S}(K,C_2;\mathcal{A})} [\![C_2]\!]^{\mathrm{tc};\#K}_{K,\mathcal{B}_K}) / (H_{K,C_2} \cup E_{K,C_2})$$
$$\parallel_{\mathcal{S}(K,C_3;\mathcal{A})} \dots \parallel_{\mathcal{S}(K,C_n;\mathcal{A})} [\![C_n]\!]^{\mathrm{tc};\#K}_{K,\mathcal{B}_K}) / (H_{K,C_n} \cup E_{K,C_n})$$

Denoted by IAQ_{K,C_j} (resp. OAQ_{K,C_j}) the parallel composition of the behaviors of the input (resp. output) asynchronous queues of K attached to C_j – whose local interactions are relabeled according to $\varphi_{K,\mathrm{async}}$ and $\varphi_{K;\mathcal{A}}$ – and by $\mathcal{OALI}^{\mathrm{input}}_{K,C_j}$ (resp. $\mathcal{OALI}^{\mathrm{output}}_{K,C_j}$) their local interactions attached to K, it turns out that $[\![K]\!]^{\mathrm{pc};\#\mathcal{B}_K}_{\mathcal{A}}$ is given by:

$$
\begin{array}{l}
IAQ_{K,C_n} \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{input}}_{K,C_n})} \\
\quad (\cdots \\
\qquad (IAQ_{K,C_2} \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{input}}_{K,C_2})} \\
\qquad\quad (IAQ_{K,C_1} \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{input}}_{K,C_1})} \\
\qquad\qquad [\![K]\!]^{\mathrm{pc;wob}}_{\mathcal{A}} \\
\qquad\quad \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{output}}_{K,C_1})} OAQ_{K,C_1}) \\
\qquad \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{output}}_{K,C_2})} OAQ_{K,C_2}) \\
\quad \cdots) \\
\parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{output}}_{K,C_n})} OAQ_{K,C_n}
\end{array}
$$

Since from the point of view of C_j and C_g the asynchronous queues of K attached to C_j are independent from the asynchronous queues of K attached to C_g for all $j \neq g$, we have that $[\![K, \mathcal{B}_K]\!]^{\mathrm{tc};\#K,\mathcal{B}_K;K}_{K,\mathcal{B}_K} \;/\; \bigcup\limits_{j=1}^{n}(H_{K,C_j} \cup E_{K,C_j})$ can be rewritten in the following way:

$$
\begin{array}{l}
(IAQ_{K,C_n} \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{input}}_{K,C_n})} \\
\quad (\cdots \\
\quad\quad (IAQ_{K,C_2} \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{input}}_{K,C_2})} \\
\quad\quad\quad (IAQ_{K,C_1} \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{input}}_{K,C_1})} \\
\quad\quad\quad\quad [\![K]\!]^{\mathrm{pc;wob}}_{\mathcal{A}} \\
\quad\quad\quad \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{output}}_{K,C_1})} OAQ_{K,C_1} \parallel_{\mathcal{S}(K,C_1;\mathcal{A})} [\![C_1]\!]^{\mathrm{tc};\#K}_{K,\mathcal{B}_K}) \,/\, (H_{K,C_1} \cup E_{K,C_1}) \\
\quad\quad \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{output}}_{K,C_2})} OAQ_{K,C_2} \parallel_{\mathcal{S}(K,C_2;\mathcal{A})} [\![C_2]\!]^{\mathrm{tc};\#K}_{K,\mathcal{B}_K}) \,/\, (H_{K,C_2} \cup E_{K,C_2}) \\
\quad \cdots) \\
\parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{output}}_{K,C_n})} OAQ_{K,C_n} \parallel_{\mathcal{S}(K,C_n;\mathcal{A})} [\![C_n]\!]^{\mathrm{tc};\#K}_{K,\mathcal{B}_K}) \,/\, (H_{K,C_n} \cup E_{K,C_n})
\end{array}
$$

Since $IAQ_{K,C_1} \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{input}}_{K,C_1})} [\![K]\!]^{\mathrm{pc;wob}}_{\mathcal{A}} \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{output}}_{K,C_1})} OAQ_{K,C_1}$ is precisely $[\![K]\!]^{\mathrm{pc};\#C_1}_{\mathcal{A}}$ and $([\![K]\!]^{\mathrm{pc};\#C_1}_{\mathcal{A}} \parallel_{\mathcal{S}(K,C_1;\mathcal{A})} [\![C_1]\!]^{\mathrm{tc};\#K}_{K,\mathcal{B}_K}) \,/\, (H_{K,C_1} \cup E_{K,C_1}) \approx_{\mathrm{B}} [\![K]\!]^{\mathrm{pc;wob}}_{\mathcal{A}}$ due to the compatibility of K with C_1, by virtue of the congruence property of $\approx_{\mathrm{B}}$ with respect to static operators – and also with respect to the context-sensitive variant of the relabeling operator governing exceptions as it encodes an injective relabeling function over actions in dot notation, which thus preserves action qualifiers – it turns out that $[\![K, \mathcal{B}_K]\!]^{\mathrm{tc};\#K,\mathcal{B}_K;K}_{K,\mathcal{B}_K} \,/\, \bigcup_{j=1}^{n} (H_{K,C_j} \cup E_{K,C_j})$ is weakly bisimilar to:

$$
\begin{array}{l}
(IAQ_{K,C_n} \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{input}}_{K,C_n})} \\
\quad (\cdots \\
\quad\quad (IAQ_{K,C_2} \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{input}}_{K,C_2})} \\
\quad\quad\quad\quad [\![K]\!]^{\mathrm{pc;wob}}_{\mathcal{A}} \\
\quad\quad \parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{output}}_{K,C_2})} OAQ_{K,C_2} \parallel_{\mathcal{S}(K,C_2;\mathcal{A})} [\![C_2]\!]^{\mathrm{tc};\#K}_{K,\mathcal{B}_K}) \,/\, (H_{K,C_2} \cup E_{K,C_2}) \\
\quad \cdots) \\
\parallel_{\varphi_{K,\mathrm{async}}(\mathcal{OALI}^{\mathrm{output}}_{K,C_n})} OAQ_{K,C_n} \parallel_{\mathcal{S}(K,C_n;\mathcal{A})} [\![C_n]\!]^{\mathrm{tc};\#K}_{K,\mathcal{B}_K}) \,/\, (H_{K,C_n} \cup E_{K,C_n})
\end{array}
$$

By reasoning in the same way for each of the other AEIs $C_2, \dots, C_n$ of $\mathcal{B}_K$, we end up with $[\![K, \mathcal{B}_K]\!]^{\mathrm{tc};\#K,\mathcal{B}_K;K}_{K,\mathcal{B}_K} \,/\, \bigcup_{j=1}^{n} (H_{K,C_j} \cup E_{K,C_j}) \approx_{\mathrm{B}} [\![K]\!]^{\mathrm{pc;wob}}_{\mathcal{A}}$. The second part of the result then follows from the fact that $\approx_{\mathrm{B}}$ preserves deadlock freedom.

(An 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.) ■

4.3. Adapting Architectural Interoperability for Cycles

A cycle is a closed simple path in the abstract enriched flow graph of an architectural description, which traverses a set $\mathcal{Y} = \{C_1, \dots, C_n\}$ of $n \geq 3$ AEIs. As explained in [1], the validity of an architectural property over a cycle cannot be investigated by analyzing the interplay between pairs of AEIs, because of the possible presence of arbitrary interferences among the various AEIs in the cycle. 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 observable behavior expected by the rest of the cycle. In other words, the observable behavior of C_j should not be altered by the insertion of C_j itself into the cycle.

In order to cope with the presence of nonsynchronous interactions, we modify the architectural interoperability check for cycles as follows.

Definition 4.8. Given an architectural description $\mathcal{A}$, let $\mathcal{Y} = \{C_1, \dots, C_n\}$ be the set of AEIs traversed by a cycle of $\mathcal{A}$, C_j be an AEI in the cycle, $H_{C_j,\mathcal{Y}}$ be the set of interactions of additional implicit AEIs of C_j that are attached to $\mathcal{Y}$, and $E_{C_j,\mathcal{Y}}$ be the set of exceptions that may be raised by semi-synchronous interactions involved in attachments between C_j and $\mathcal{Y}$. We say that C_j interoperates with the other AEIs in the cycle iff:

$$[\![\mathcal{Y}]\!]_{\mathcal{A}}^{\mathrm{tc};\#\mathcal{Y};C_j} \,/\, (\mathit{Name} - \mathcal{V}_{C_j;\mathcal{A}}) \,/\, (H_{C_j,\mathcal{Y}} \cup E_{C_j,\mathcal{Y}}) \ \approx_{\mathrm{B}} \ [\![C_j]\!]_{\mathcal{A}}^{\mathrm{pc;wob}} \qquad \blacksquare$$

All possible originally asynchronous local interactions of the other AEIs in the cycle and all of their interactions that are not attached to C_j have been hidden by taking the totally closed interacting semantics of those AEIs and by leaving visible only the actions in $\mathcal{V}_{C_j;\mathcal{A}}$. We also observe that, whenever C_j has no local nonsynchronous interactions and is not attached to semi-synchronous interactions of other AEIs in the cycle, then $H_{C_j,\mathcal{Y}} \cup E_{C_j,\mathcal{Y}} = \emptyset$ and both $[\![C_j]\!]_{\mathcal{A}}^{\mathrm{pc};\#\mathcal{Y}}$ and $[\![C_j]\!]_{\mathcal{A}}^{\mathrm{pc;wob}}$ coincide with $[\![C_j]\!]_{\mathcal{A}}^{\mathrm{tc;wob}}$.

We now extend the interoperability theorem of [1] to nonsynchronous interactions. This provides a sufficient condition for reducing the deadlock verification of the entire cycle to the deadlock verification of one of its AEIs.

Theorem 4.9. Let $\mathcal{A}$, $\mathcal{Y}$, C_j, $H_{C_j,\mathcal{Y}}$, and $E_{C_j,\mathcal{Y}}$ be the same as Def. 4.8. Whenever C_j interoperates with the other AEIs in the cycle, then $[\![\mathcal{Y}]\!]_{\mathcal{A}}^{\mathrm{tc};\#\mathcal{Y};C_j} \,/\, (\mathit{Name} - \mathcal{V}_{C_j;\mathcal{A}}) \,/\, (H_{C_j,\mathcal{Y}} \cup E_{C_j,\mathcal{Y}})$ is deadlock free iff so is $[\![C_j]\!]_{\mathcal{A}}^{\mathrm{pc;wob}}$.

Proof A straightforward consequence of Def. 4.8 and of the fact that $\approx_{\mathrm{B}}$ preserves deadlock freedom. $\blacksquare$

4.4. Case Study: An Applet-Based Simulator for a Cruise Control System

In this section, we discuss the application of the modified architectural checks by revisiting the cruise control system considered in [32, 11].

Once the engine has been turned on, this system is governed by the two standard pedals of the automobile – accelerator and brake – and by three additional 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.

Let us describe the cruise control system with PADL. The sensor AET is defined as follows:

```
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  SYNC UNI detected_engine_on; detected_engine_off;
                               detected_accelerator; detected_brake;
                               detected_on; detected_off; detected_resume
  OUTPUT_INTERACTIONS SYNC UNI press_accelerator; press_brake;
                               press_on; press_off; press_resume
                           AND turn_engine_on; turn_engine_off
```

The speed controller triggers the speed actuator on the basis of the commands forwarded by the sensor. It can be 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), or suspended (after pressing any pedal or button different from on/resume in the cruising state):

```
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  SYNC UNI turned_engine_on; turned_engine_off;
                               pressed_accelerator; pressed_brake;
                               pressed_on; pressed_off; pressed_resume
  OUTPUT_INTERACTIONS SYNC UNI trigger_record; trigger_resume;
                               trigger_disable
```

The speed detector periodically communicates the number of wheel revolutions per time unit to the speed actuator:

```
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  SYNC UNI turned_engine_on; turned_engine_off
  OUTPUT_INTERACTIONS SYNC 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 disabled (until the on/resume button is pressed) or enabled (until any pedal or button different from on/resume is pressed):

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

Suppose we want to design an applet-based simulator for the cruise control system. The applet must contain a panel with seven software buttons – corresponding to turning the engine on/off, the two pedals, and the three hardware buttons – and 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 panel 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 panel cannot interact with the sensor, rather it emits a beep.

In order not to block the simulator when the pressure of a software button fails, we need to model several operations of the panel through semi-synchronous interactions, as shown below:

```
ARCHI_ELEM_TYPE Panel_Type(void)
  BEHAVIOR
    Unallocated(void; void) =
      init_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 init_applet; start_applet;
                              stop_applet; destroy_applet
  OUTPUT_INTERACTIONS SSYNC UNI signal_engine_on; signal_engine_off;
                              signal_accelerator; signal_brake;
                              signal_on; signal_off; signal_resume
```

where all the input interactions are related to user commands for starting/stopping the simulator.

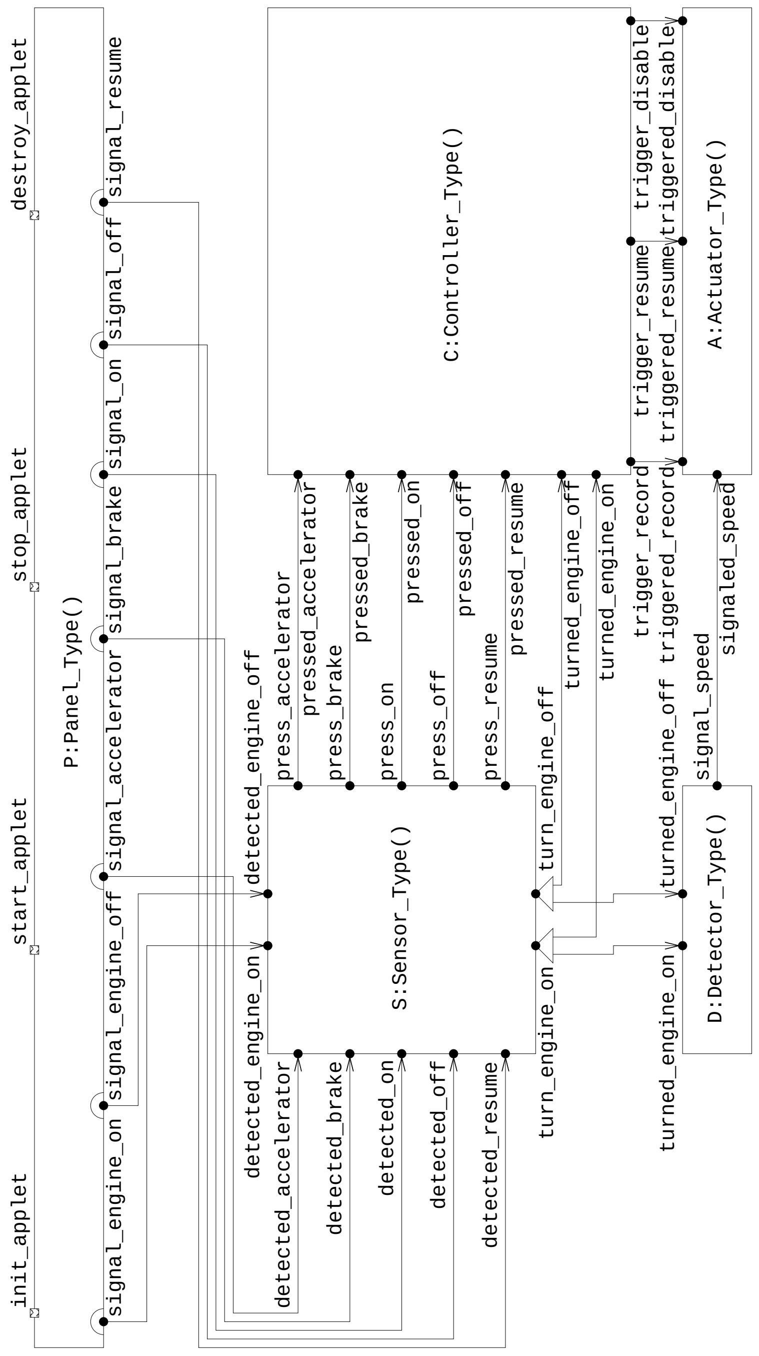


Figure 3: Enriched flow graph of the cruise control system simulator

The topology of the overall system is described through the enriched flow graph of Fig. 3. As can be noted, it is an intersection of a cycle and a star, where the cycle traverses AEIs `S`, `C`, `D`, and `A`.

Suppose we wish to verify whether the applet-based simulator is deadlock free. First of all, we observe that all the AEIs are deadlock free. Then, we consider the cycle and we apply to it the interoperability check. Since there are no nonsynchronous interactions within the cycle, applying the modified interoperability check to the cycle boils down to applying the original check. The outcome is thus known from [11]: `S` interoperates with `C`, `D`, and `A`, hence the cycle can be reduced to $[\![\mathtt{S}]\!]^{\mathrm{pc;wob}}_{\mathtt{P,S,C,D,A}}$ (Thm. 4.9).

Now, it is easy to see that `S` is compatible with the only AEI in the only acyclic portion of the topology – `P` – because all the exceptions that the semi-synchronous interactions of `P` may raise are hidden when applying the check. Therefore, we can conclude that the entire architectural description can be reduced to $[\![\mathtt{S}]\!]^{\mathrm{pc;wob}}_{\mathtt{P,S,C,D,A}}$, and hence it is deadlock free because so is $[\![\mathtt{S}]\!]^{\mathrm{pc;wob}}_{\mathtt{P,S,C,D,A}}$ (Thm. 4.7).

5. Code Generation

The architectural description of a software system should be used not only for modeling and verification purposes, but also for guiding subsequent stages of the software development process. In particular, it should serve as a basis for synthesizing the software system itself. As an attempt to bridge the gap between system modeling/verification and system implementation, we propose an approach that automatically generates multithreaded Java software from PADL descriptions containing an arbitrary combination of synchronous, semi-synchronous, and asynchronous interactions.

Similar to [32], the choice of Java as target language is made for two reasons. First, Java supports multithreading and offers a set of mechanisms for the well-structured management of threads and their shared data, which should simplify the code generation task given the concurrency inherent in process algebraic architectural descriptions. Second, its object-oriented nature – and specifically its encapsulation capability – makes Java an appropriate candidate for coping with the high level of abstraction typical of process algebraic architectural description languages. In fact, as can be expected, the translation of PADL descriptions into Java software cannot be complete, and hence will require the intervention of the software developer in specific positions of the generated code, e.g., for inserting the Java statements corresponding to internal actions.

Another good reason for selecting Java is that it is equipped with software model-checking tools – like, e.g., Java PathFinder [39] – that complement the analysis conducted on PADL descriptions by making it possible the verification of property preservation at the code level. In fact, although property preservation is guaranteed under certain constraints as we will see, an inappropriate intervention of the software developer on the generated code may lead to the violation of properties proved at the architectural level.

In order to compare our approach with related work for automatically generating code from architectural descriptions, it is worth recalling two families distinguished on the basis of the distance between the formalism used for describing software architectures (in our case, PADL) and the implementation language in which code is generated (in our case, Java). The first family, characterized by an exogenous transformation, is the long-distance one. In this family, the formalism is kept well separated from the implementation language and descriptions are entirely translated into code. To this family belong architectural description languages endowed with code generation facilities like Aesop [24], C2SADEL [33], and Darwin [31]. The second family, characterized by a semi-endogenous transformation, is the short-distance one. In this family, the formalism is embedded in the implementation code in form of special comments, as in SyncGen [21], or in form of special keywords and statements, as in ArchJava [2]. In this case, only special symbols are translated into implementation code, while the rest is left unchanged.

While the latter transformation can offer a flexible support to software developers – as classical programming techniques and patterns can be applied when designing systems – the level of abstraction of the underlying architectural formalism is usually low. In the case of process algebraic architectural description languages, the transformation typically adopted for generating code is the former. The reason is that such languages are specifically conceived for abstracting high-level properties of entire software systems, hence they are distant from implementation languages.

One of the ideas at the basis of our approach is the provision of a library of software components – a Java package called `Sync` – for adding architectural capabilities to the target programming language, in order to shorten the distance from the architectural description language from which the code will be generated. Hence, our approach can be viewed as a semi-exogenous transformation, as opposed to the semi-endogenous one. Based on the classification given in [20], we can say that if the target model is considered as a text, a semi-endogenous transformation can be simply realized as a model-to-text, template-based transformation. Unfortunately, this transformation cannot be applied with the same simplicity to our semi-exogenous approach, as the distance from model to text is still long. However, thanks to the presence of a package like `Sync`, a model-to-text transformation can be implemented very easily based on the pattern Visitor [22].

In this section, we present the two phases of our approach by illustrating them on the same architectural description of an applet-based simulator for a cruise control system as the previous section. In the first phase, we develop an architecture-driven technique for thread coordination management based on package `Sync` (Sect. 5.1). In the second phase, we handle the translation of the process-algebraically-specified behavior of individual software components into threads (Sect. 5.2). Then, we focus on the preservation at the code level of the properties proved at the architectural level (Sect. 5.3).

5.1. First Phase: Thread Coordination Management

The first phase of our approach to the generation of multithreaded Java code from PADL descriptions deals with the thread coordination management. This is accomplished by developing a Java package called `Sync`, which automatically takes care of the details of thread synchronization. Both the implementation of the package and the use of its units for coordinating threads are guided by architecture-level abstractions.

In the following, we present a reference thread communication model and then we illustrate the structure and the usage of `Sync`.

5.1.1. Thread Communication Model

The thread communication model adopted by package `Sync`, which fully complies with the interaction qualifiers of PADL, relies on two roles (sender and receiver) and encompasses two different dimensions.

The first dimension – thread communication synchronicity – comprises nine values arising from all possible combinations of a synchronous, semi-synchronous, or asynchronous activity on the sender side with a synchronous, semi-synchronous, or asynchronous activity on the receiver side. Similar to PADL, in a synchronous-to-synchronous communication both threads wait for the other to become ready. In the semi-synchronous case, the thread checks whether the other is ready and, if not, raises an exception without blocking. In the asynchronous case, the thread simply sends/receives a signal or message through a buffer and then proceeds independently of the status of the other thread (an exception is raised at the asynchronous receiving side if no signal or message is available in the buffer).

The second dimension – thread communication multiplicity – comprises the following five values: uni-to-uni, and-to-uni, uni-to-and, or-to-uni, uni-to-or. As with PADL, in a uni-to-uni communication only two threads are involved (unicast). In an and-to-uni/uni-to-and communication, a thread simultaneously communicates with several other threads (multicast). Finally, in an or-to-uni/uni-to-or communication a thread communicates with only one thread selected out of a set of other threads (server-clients).

5.1.2. Structure of Package `Sync`

The Java package `Sync`, which adheres to the above mentioned communication model, is structured into four conceptual layers: `Connector`, `Port`, `RunnableElem`, and `RunnableArchi`. Each of them corresponds to a different architectural abstraction and comprises a set of components realized through Java classes and interfaces, some of which are visible by the software developer. The related class diagrams are shown in Fig. 4.

The first layer, called `Connector`, is a set of invisible Java classes and interfaces that are used within `Sync` to perform synchronizations and data transfers between two threads. As sketched in Fig. 4, there are nine classes realizing – consistently with the adopted thread communication model – the nine combinations of communication synchronicities.

This layer is inspired by the traditional producer-consumer model, where the producer and the consumer represent the sender and receiver threads, respectively. This is implemented by equipping every `Connector` class with a buffer shared only by the two related threads. Each such class also has two interface methods – `send()` and `receive()` – for accessing the buffer in a mutually exclusive way and two interface methods – `obsSnd()` and `obsRcv()` – for observing the status of the threads using a `Connector` object in the case of nonasynchronous and-/or-interactions.

The second layer, called `Port`, is a set of Java classes and interfaces that realize the abstraction corresponding to a set of statements through which a thread interacts with possibly many other threads. As sketched in Fig. 4, there are eighteen classes combining – through `Connector` objects – the three cases of communication multiplicity with the three cases of communication synchronicity both on the receiver side and on the sender side.

This layer gives rise to suitable objects, each of which will be attached

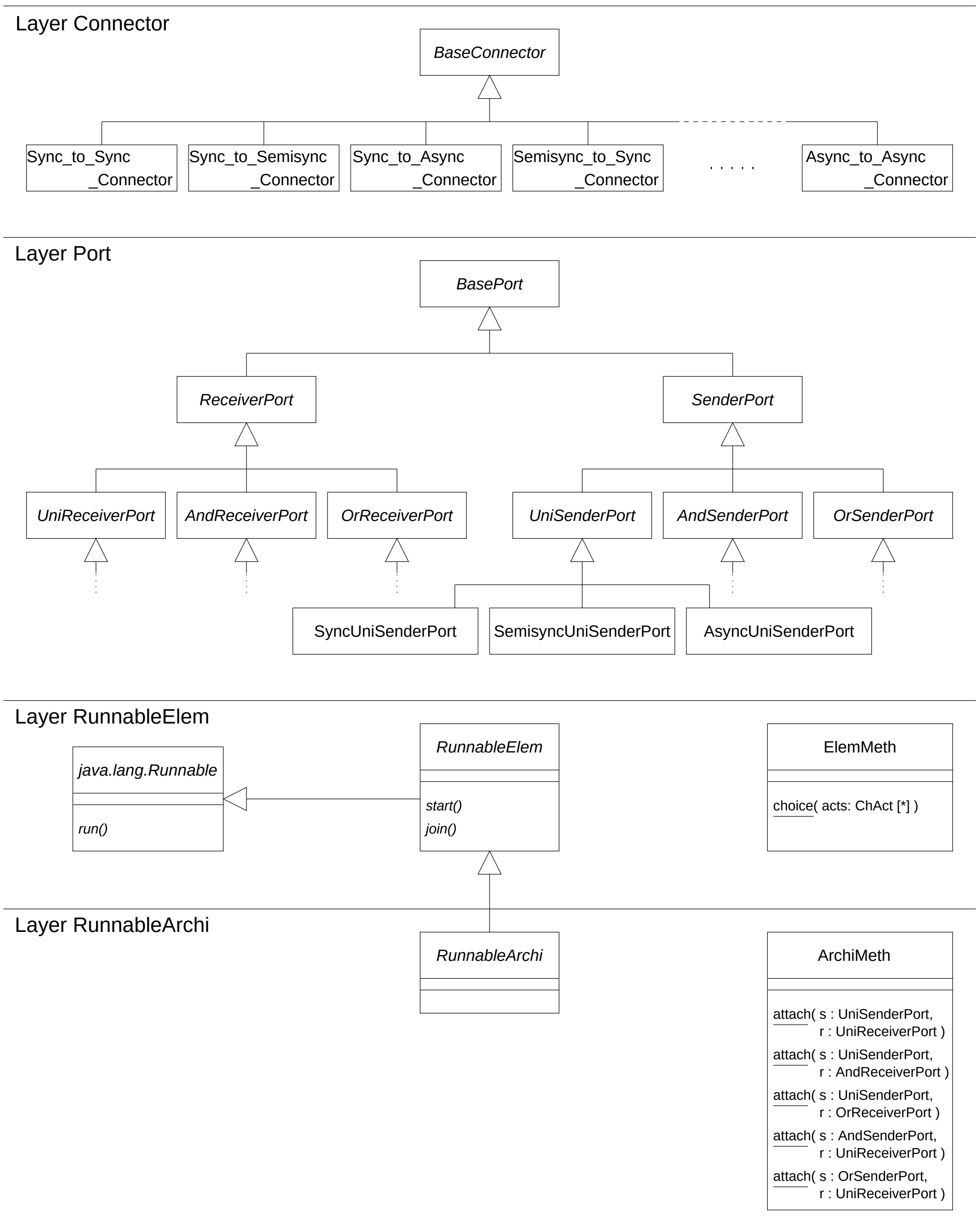


Figure 4: Class diagrams of the conceptual layers of `Sync`

either to a single Connector object – if representing a uni-interaction – or to several Connector objects – if representing an and-/or-interaction. Every Port object must be instantiated by specifying its owner thread in the Port constructor. Each of the nine Port classes related to the sender side is equipped with a send() method, whereas each of the nine Port classes related to the receiver side is equipped with a receive() method. Each of the two methods makes use of the homonymous method of the associated Connector objects and raises an UnattachedPortException whenever the Port object is not connected to any other Port object. This happens if the interaction associated with the Port object is an architectural interaction not attached to any other interaction. A different exception, SemisyncPortNotReadyException, is raised if the associated interaction is semi-synchronous and the connected Port objects are not ready to communicate. Both exceptions are defined as derived classes of a base class called SyncException defined within Sync.

The third layer, called RunnableElem, is an interface derived from the standard Runnable interface that realizes the abstraction corresponding to a thread in a concurrent Java program. This layer provides support for the generation of Java threads from the process-algebraically-specified behavior of AETs in a PADL description. In particular, as shown in Fig. 4, it is also equipped with a class called ElemMeth, which defines a static method called choice() used for the translation of alternative compositions.

The fourth layer, called RunnableArchi, is a RunnableElem-derived interface that realizes the abstraction corresponding to a concurrent Java program. The compatibility of RunnableArchi with RunnableElem provides support for hierarchical software development. As shown in Fig. 4, in order to avoid the direct handling of Connector objects, this layer is also equipped with a class called ArchiMeth, which defines a family of five static methods called attach(). Each of them receives two parameters, which must be a sender Port object and a receiver Port object, and connects them by creating a Connector object only if they refer to two different owner threads according to one of the five combinations of communication multiplicities admitted by the adopted thread communication model.

5.1.3. Usage of Package Sync

From a code generation viewpoint, package Sync is a repository of architectural abstractions ensuring the correct and transparent handling of synchronizations and data exchanges among the threads of the software system being designed. The structure of the generated code is shown in Fig. 5.

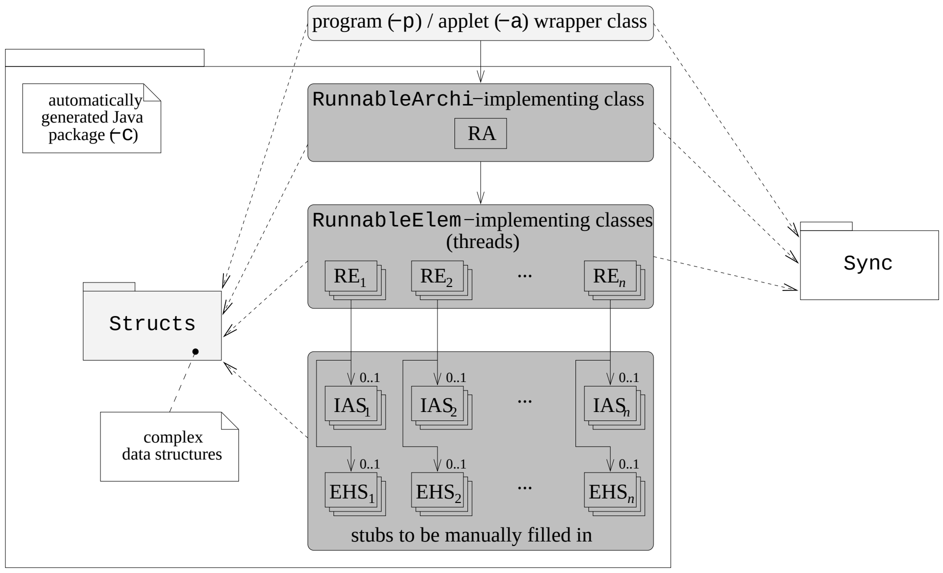

program (-p) / applet (-a) wrapper class
automatically generated Java package (-c)
RunnableArchi-implementing class
RA
RunnableElem-implementing classes (threads)
RE1
RE2
···
REn
Sync
Structs
complex data structures
0..1
IAS1
IAS2
IASn
EHS1
EHS2
EHSn
stubs to be manually filled in

Figure 5: Structure of the `Sync`-based generated code

Starting from a PADL description of the system, it is necessary to create a `RunnableArchi` object for the overall system, together with as many `RunnableElem` objects as there are AEIs in the PADL description (see the upper part of Fig. 5). Then, it is necessary to create as many `Port` objects as there are interactions in the AEIs synthesized as `RunnableElem` objects. Afterwards, it is necessary to create all the `Connector` objects that are needed to make the AEIs synthesized as `RunnableElem` objects communicate – according to the attachments in the PADL description – through their interactions synthesized as `Port` objects.

The lower part of Fig. 5 shows stub classes to be manually filled in. As we will see in the second phase, these stubs are generated for managing the internal actions of the AETs (IAS) and for handling exceptions that may be raised by the interactions of the instances of those AETs (EHS) – i.e., `UnattachedPortException` and `SemisyncPortNotReadyException`.

Fig. 5 also shows a package called `Structs`. The reason is that most of the data types provided by PADL – such as boolean, (bounded) integer, real, list, and array – can be trivially translated into built-in Java data types. However, the record data type and the generic object data type provided by PADL need ad-hoc classes and interfaces made available through `Structs`.

Each of the classes generated by PADL2Java for a given PADL description is stored into a distinct `.java` file. All these files are contained in a single package with the same name as the PADL description source file. Further `.java` files may be generated depending on the specified translation option.

Fig. 5 shows the various options that are available at code generation time: `-c`, `-p`, and `-a`. The default option `-c` stands for the generation of the package depicted in the figure, with no additional classes. Option `-p` stands for the synthesis of a full Java program. This can be achieved through the generation of a further public class that contains only method `main()`, which acts as a wrapper for the `RunnableArchi`-implementing class. Option `-a` stands for the synthesis of a Java applet. This can be achieved through the generation of a further public class derived from the standard `JApplet` class, which gives rise to a wrapper for the `RunnableArchi`-implementing class.

Both for `-p` and for `-a`, the wrapper class creates an instance of the `RunnableArchi`-implementing class and declares a `SyncUniSenderPort` or `SyncUniReceiverPort` object for each architectural interaction declared in the `RunnableArchi`-implementing class. In this way, the wrapper is allowed to dispatch suitable commands to the right `RunnableElem` objects whenever necessary. Moreover, in the `-a` case the wrapper class defines a number of stubs related to the initialization, activation, deactivation, and destruction of the applet, which will be filled in by the software developer if needed.

Example 5.1. Consider the cruise control system simulator of Sect. 4.4 and suppose we want to synthesize it as multithreaded Java software from its PADL description. Since it is an applet-based simulator, we need to generate an applet wrapper. The following `.java` file contains the `JApplet`-derived class synthesized for the wrapper, where `Cruise_Control_System_package` is the package for the cruise control system:

```
import Sync.*;
import Cruise_Control_System_package.Cruise_Control_System;
import javax.swing.JApplet;

public class MainAppletClass extends JApplet {

  //---------- DECLARING ARCHITECTURE -----------//
  Cruise_Control_System archiInstance = null;

  //---------- DECLARING APPLET PORTS -----------//
  SyncUniSenderPort to_P_init_applet;
  SyncUniSenderPort to_P_start_applet;
  SyncUniSenderPort to_P_stop_applet;
  SyncUniSenderPort to_P_destroy_applet;

  //---------- DEFINING APPLET MEMBERS ----------//
  public void init() {
    if (archiInstance == null) {
      archiInstance = new Cruise_Control_System();
      // to_P_init_applet = new SyncUniSenderPort(this);
      // to_P_start_applet = new SyncUniSenderPort(this);
      // to_P_stop_applet = new SyncUniSenderPort(this);
      // to_P_destroy_applet = new SyncUniSenderPort(this);
      // try {
      //   ArchiMeth.attach(to_P_init_applet,
      //                    archiInstance.P_init_applet);
      //   ArchiMeth.attach(to_P_start_applet,
      //                    archiInstance.P_start_applet);
      //   ArchiMeth.attach(to_P_stop_applet,
      //                    archiInstance.P_stop_applet);
      //   ArchiMeth.attach(to_P_destroy_applet,
      //                    archiInstance.P_destroy_applet);
      // } catch(BadAttachmentException e) {}
      archiInstance.start();
      // FILL IN THE FINAL PART OF THE METHOD BODY IF NEEDED
  }

  public void start() {
    // FILL IN THE METHOD BODY IF NEEDED
  }

  public void stop() {
    // FILL IN THE METHOD BODY IF NEEDED
  }

  public void destroy() {
    // FILL IN THE INITIAL PART OF THE METHOD BODY IF NEEDED
    archiInstance = null;
  }
}
```

As can be noted, for each of the four architectural input interaction of AEI `P`, we have the declaration of a sender port whose name coincides with the name of the corresponding architectural input interaction prefixed by `to_` (the prefix is `from_` in the case of an architectural output interaction). Each such sender port will manage the commands given by the user of the applet-based simulator when initializing, starting, stopping, or destroying the applet-based simulator itself depending on how the software developer fills in the body of the methods associated with those operations.

As far as method `init()` is concerned, the initial part of its body includes some commented statements for instantiating the applet ports and attaching them to the corresponding architectural interactions. Those statements are commented because, in general, it is not necessarily the case that all the architectural interactions must be attached to the corresponding ports in the wrapper. For instance, some of them should be attached to other software units. It is the responsibility of the software developer to uncomment the statements that are appropriate for the specific system. ■

5.1.4. Structure of the `RunnableArchi`*-Implementing Class*

When using package `Sync` in the translation process, the Java class synthesized to implement the `RunnableArchi` interface for the overall system has the same name as the architectural type and is structured as follows:

```
public class ◁architectural type name▷ implements RunnableArchi {
  ◁Declaring Runnable Elements▷
  ◁Declaring Architectural Interactions▷
  ◁Defining Constructor▷
  ◁Building Architecture▷ :
    ◁Instantiating Runnable Elements▷
    ◁Assigning Architectural Interactions▷
    ◁Attaching Local Interactions▷
  ◁Running Architecture▷
}
```

The first section, *Declaring Runnable Elements*, declares an object of a `RunnableElem`-implementing class – without instantiating it – for each AEI declared in the PADL description.

The second section, *Declaring Architectural Interactions*, declares a public `Port` object for each architectural interaction declared in the PADL description. Such objects are public because the architectural interactions are the interfaces of the whole system, hence support must be provided for them to be used for the hierarchical or compositional modeling of complex systems as well as within wrapper classes.

The third section, *Defining Constructor*, defines the class constructor together with its parameters, which coincide with the parameters of the architectural type.

The fourth section, *Building Architecture*, defines a method called `buildArchiTopology()` that constructs the architecture topology in a way similar to the architectural topology section of the PADL description. First, it instantiates the previously declared `RunnableElem` objects. Second, it assigns the previously declared public `Port` objects through the corresponding `Port` objects of the newly instantiated `RunnableElem` objects. Third, it invokes the method `attach()` defined within `Sync` in order to connect the `Port` objects of the newly instantiated `RunnableElem` objects according to the attachments declared in the PADL description.

Finally, the fifth section, *Running Architecture*, declares the thread associated with the class itself. Moreover, it defines the public methods `start()`, which starts the previously declared thread, `join()`, which allows other threads to wait for the end of the execution of the previously declared thread, and `run()`, which starts all the previously instantiated `RunnableElem` objects and then waits for their termination.

Example 5.2. We now continue Ex. 5.1. The following `.java` file contains the `RunnableArchi` interface for the cruise control system:

```
package Cruise_Control_System_package;

import Sync.*;

public class Cruise_Control_System implements RunnableArchi {

  //-------- DECLARING RUNNABLE ELEMENTS --------//
  Panel_Type P;
  Sensor_Type S;
  Controller_Type C;
  Detector_Type D;
  Actuator_Type A;

  //--- DECLARING ARCHITECTURAL INTERACTIONS ----//
  public SyncUniReceiverPort P_init_applet;
  public SyncUniReceiverPort P_start_applet;
  public SyncUniReceiverPort P_stop_applet;
  public SyncUniReceiverPort P_destroy_applet;

  //----------- DEFINING CONSTRUCTOR ------------//
  Cruise_Control_System() {
    buildArchiTopology();
  }
```

```
//---------- BUILDING ARCHITECTURE ----------//
void buildArchiTopology() {

  // INSTANTIATING RUNNABLE ELEMENTS:
  P = new Panel_Type();
  S = new Sensor_Type();
  C = new Controller_Type();
  D = new Detector_Type();
  A = new Actuator_Type();

  // ASSIGNING ARCHITECTURAL INTERACTIONS:
  this.P_init_applet = P.init_applet;
  this.P_start_applet = P.start_applet;
  this.P_stop_applet = P.stop_applet;
  this.P_destroy_applet = P.destroy_applet;

  // ATTACHING LOCAL INTERACTIONS:
  try {
    ArchiMeth.attach(P.signal_engine_on,
                     S.detected_engine_on);
    ArchiMeth.attach(P.signal_engine_off,
                     S.detected_engine_off);
    ArchiMeth.attach(P.signal_accelerator,
                     S.detected_accelerator);
    ArchiMeth.attach(P.signal_brake,
                     S.detected_brake);
    ArchiMeth.attach(P.signal_on,
                     S.detected_on);
    ArchiMeth.attach(P.signal_off,
                     S.detected_off);
    ArchiMeth.attach(P.signal_resume,
                     S.detected_resume);
    ArchiMeth.attach(S.turn_engine_on,
                     C.turned_engine_on);
    ArchiMeth.attach(S.turn_engine_on,
                     D.turned_engine_on);
    ArchiMeth.attach(S.turn_engine_off,
                     C.turned_engine_off);
    ArchiMeth.attach(S.turn_engine_off,
                     D.turned_engine_off);
    ArchiMeth.attach(S.press_accelerator,
                     C.pressed_accelerator);
    ArchiMeth.attach(S.press_brake,
                     C.pressed_brake);
```

```
      ArchiMeth.attach(S.press_on,
                       C.pressed_on);
      ArchiMeth.attach(S.press_off,
                       C.pressed_off);
      ArchiMeth.attach(S.press_resume,
                       C.pressed_resume);
      ArchiMeth.attach(C.trigger_record,
                       A.triggered_record);
      ArchiMeth.attach(C.trigger_resume,
                       A.triggered_resume);
      ArchiMeth.attach(C.trigger_disable,
                       A.triggered_disable);
      ArchiMeth.attach(D.signal_speed,
                       A.signaled_speed);
    } catch(BadAttachmentException e) {}
  }

  //----------- RUNNING ARCHITECTURE ------------//
  Thread th_Cruise_Control_System = null;

  public void start() {
    (th_Cruise_Control_System = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Cruise_Control_System.join();
  }

  public void run() {
    P.start();
    S.start();
    C.start();
    D.start();
    A.start();
    try {
      P.join();
      S.join();
      C.join();
      D.join();
      A.join();
    } catch(InterruptedException e) {}
  }
}
```

Note that it faithfully implements the topology of Fig. 3. ■

5.1.5. Structure of `RunnableElem`*-Implementing Classes*

Similar to the case of `RunnableArchi`, each of the Java classes synthesized to implement the `RunnableElem` interface corresponding to an AET has the same name as the AET and is structured as follows:

```
class ◁architectural element type name▷ implements RunnableElem {
  ◁Declaring Behavioral Equations Interfaces▷
  ◁Instantiating Interactions▷
  ◁Declaring Stubs▷
  ◁Defining Constructor▷
  ◁Defining Behavior▷
  ◁Running Element▷
}
```

The first section, *Declaring Behavioral Equations Interfaces*, defines an interface called `BehavioralEquationInterface` and declares an equation object of such an interface for each behavioral equation occurring in the AET definition.

The second section, *Instantiating Interactions*, instantiates various `Port` objects of various types (see the second layer of Fig. 4) on the basis of the interactions occurring in the AET definition and their qualifiers.

The third section, *Declaring Stubs*, declares two stub objects to be manually filled in later on. As already mentioned, one stub object is for the translation of the internal actions occurring in the AET definition, while the other stub is for handling exceptions related to the interactions occurring in the AET definition.

The fourth section, *Defining Constructor*, defines the class constructor together with its parameters, which coincide with the parameters of the AET. The constructor declares the parameters as nonpublic members in order to store their values and make them available throughout the thread execution. Then, the constructor invokes the method defined in the next section.

The fifth section, *Defining Behavior*, creates instances of anonymous classes implementing `BehavioralEquationInterface` and assigns them to the previously mentioned equation objects. Each anonymous class translates a different behavioral equation occurring in the AET. The only method declared by the interface, `behavEqCall()`, is defined here for each equation object and will be discussed in the second phase.

Finally, the sixth section, *Running Element*, declares the thread associated with the class itself. Moreover, it defines the public methods `start()`, which starts the previously declared thread, `join()`, which allows other threads to wait for the end of the execution of the previously declared thread, and `run()`, which instantiates the two stub classes and executes the various equation objects.

Example 5.3. We now continue Ex. 5.1 and Ex. 5.2. The following `.java` file contains the `RunnableElem` interface for `Panel_Type`:

```
package Cruise_Control_System_package;

import Sync.*;

class Panel_Type implements RunnableElem {

  //- DECLARING BEHAVIORAL EQUATIONS INTERFACES -//
  interface BehavioralEquationInterface { void behavEqCall(); }
  BehavioralEquationInterface Unallocated,
                              Active,
                              Checking,
                              Inactive;
  BehavioralEquationInterface nextBehavEq;
  Object[] actualPars;

  //-------- INSTANTIATING INTERACTIONS ---------//
  SyncUniReceiverPort init_applet = new SyncUniReceiverPort(this);
  SyncUniReceiverPort start_applet = new SyncUniReceiverPort(this);
  SyncUniReceiverPort stop_applet = new SyncUniReceiverPort(this);
  SyncUniReceiverPort destroy_applet = new SyncUniReceiverPort(this);
  SemisyncUniSenderPort signal_engine_on = new SemisyncUniSenderPort(this);
  SemisyncUniSenderPort signal_engine_off = new SemisyncUniSenderPort(this);
  SemisyncUniSenderPort signal_accelerator = new SemisyncUniSenderPort(this);
  SemisyncUniSenderPort signal_brake = new SemisyncUniSenderPort(this);
  SemisyncUniSenderPort signal_on = new SemisyncUniSenderPort(this);
  SemisyncUniSenderPort signal_off = new SemisyncUniSenderPort(this);
  SemisyncUniSenderPort signal_resume = new SemisyncUniSenderPort(this);

  //-------------- DECLARING STUBS --------------//
  IAS_Panel_Type internal_Panel_Type;
  EHS_Panel_Type exception_Panel_Type;

  //----------- DEFINING CONSTRUCTOR ------------//
  Panel_Type() {
    defineBehavEquations();
  }
  //------------ DEFINING BEHAVIOR -------------//
  void defineBehavEquations() {
    ...
  }

  //------------- RUNNING ELEMENT --------------//
  Thread th_Panel_Type = null;

  public void start() {
    (th_Panel_Type = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Panel_Type.join();
  }

  public void run() {
    ...
  }
}
```

As can be noted, the definitions of methods `defineBehavEquations()` and `run()` have been omitted because they are strictly related to the synthesis of the thread behavior. Therefore, they will be shown after presenting the second phase of our approach. ■

5.2. Second Phase: Thread Behavior Generation

The second phase of our approach to the generation of multithreaded Java code from PADL descriptions deals with the translation of the process algebraic specification of the behavior of the AETs into thread classes. Due to the different level of abstraction of an architectural description language and of a programming language, only a partial translation based on stubs is possible, with the preservation of architectural properties depending on the way in which the stubs are filled in by the software developer.

In the following, we introduce a reference thread generation model and we show how to synthesize thread method `run()` and how to translate process algebraic operators consistent with the adopted model in order to synthesize method `defineBehavEquations()`. We also show how to synthesize stubs for handling exceptions and internal actions.

5.2.1. Thread Generation Model

A finite state machine model is adopted to guide the generation of a thread from the description of an AET made out of a sequence of process algebraic defining equations. Large conditional statements or table-based approaches do not guarantee high efficiency when many conditions or associations have to be checked at run time for each invocation of a behavioral equation. Therefore, we generate code on the basis of the behavioral pattern State [22].

As shown in Sect. 5.1.5, each AET is translated into a class implementing the interface `RunnableElem`. Each state of the behavior of the AET is defined as an inner class implementing the interface `BehavioralEquationInterface` and defining the method `behavEqCall()`. The idea is that every behavioral equation is translated into an inner state class having the same name as the equation. The code for an inner state class is generated by proceeding by induction on the syntactical structure – `stop`, behavioral invocation, action prefix, and `choice` – of the process algebraic term occurring on the right-hand side of the corresponding behavioral equation.

The class associated with an AET also defines the member `nextBehavEq`, a reference to an object implementing `BehavioralEquationInterface`, and the member `actualPars`, a reference to an array of objects. These references, which are shared and visible by all the inner state classes, are in charge of defining the state transitions, i.e., the next behavioral equation to be executed and the actual parameters to be passed.

We conclude by observing that a different treatment is needed for action prefixes depending on whether the related actions are interactions or internal actions. An interaction is involved in communications, hence it is automatically managed via package `Sync`. However, as we have seen in Sect. 5.1.1, the execution of an interaction may result in two kinds of exception, whose handling is left to the software developer. Our thread generation model thus includes the possibility of associating an exception handling stub (EHS) with each interaction, to be filled in by the software developer. By contrast, an internal action is not involved at all in communications, as it takes place inside an AET. In this case, the architectural description does not provide any information about how to translate the action into a sequence of Java statements. As a consequence, our thread generation model also includes the possibility of associating an internal action stub (IAS) with each internal action, to be filled in by the software developer.

5.2.2. Synthesizing Thread Method `run()`

The class corresponding to an AET comprises the constructor and method `run()`. While the former instantiates the inner state classes by calling method `defineBehavEquations()`, as shown below the latter first of all instantiates the EHSs and the IASs declared as members of the class:

```
public void run() {
  ◁EHS instantiation▷
  ◁IAS instantiation▷
  nextBehavEq = ◁first behavioral equation▷;
  actualPars = ◁actual parameters of the first behavioral equation▷;
  while (nextBehavEq != null)
    nextBehavEq.behavEqCall();
}
```

Method `run()` then assigns the state class instance representing the first behavioral equation to `nextBehavEq` and the related parameters to `actualPars`. A `while` statement carries out the execution of the behavioral equations starting from the first one, by repeatedly invoking method `behavEqCall()` on the state class instance referenced by `nextBehavEq`. When `nextBehavEq` is set to `null`, the execution of method `run()` terminates.

Example 5.4. We complete the second omitted part of Ex. 5.3 by showing the definition of method `run()` for the class associated with `Panel_Type`:

```
public void run() {
  internal_Panel_Type = new IAS_Panel_Type();
  exception_Panel_Type = new EHS_Panel_Type();
  nextBehavEq = Unallocated;
  actualPars = null;
  while (nextBehavEq != null)
    nextBehavEq.behavEqCall();
}
```

■

5.2.3. Translating `stop`

Process term `stop` represents the situation in which no further action can be executed. It is therefore translated by assigning `null` both to `nextBehavEq` and to `actualPars`. As a consequence, when encountering `stop`, method `run()` terminates its execution.

5.2.4. Translating Behavioral Invocations

The behavioral invocation $B(\underline{e})$ represents a process term that behaves as the behavioral equation whose identifier is B, when passing the possibly empty sequence of actual parameters $\underline{e}$. A behavioral invocation, which can occur only within the scope of an action prefix operator, is not translated into a method call, as this may result in the generation of inefficient code in case of recursion. Instead, a behavioral invocation is translated into an assignment to `nextBehavEq` of an instance of the inner state class that corresponds to the next behavioral equation, followed by an assigment to `actualPars` of the actual parameters needed by the next behavioral equation.

5.2.5. Translating Action Prefixes

The action prefix operator is used to represent a process term that can execute an action and then behaves as described by another process term. As already mentioned, the translation of the action depends on whether it is an interaction or an internal action.

In the first case, the action is translated into an invocation of method `send()` – if it is an output interaction – or method `receive()` – if it is an input interaction – of the corresponding `Port` object. If the interaction is semi-synchronous or architectural, its translation must then be completed by filling in the related method in an EHS.

In the second case, the action translation is completely left to the software developer, as internal actions cannot be treated automatically at all. A method for each of them is placed in a distinct IAS, which has to be filled in by the software developer with the corresponding Java statements. As a consequence, every occurrence of an internal action is translated into an invocation of the related method in an IAS.

Example 5.5. We now continue Ex. 5.1, Ex. 5.2, and Ex. 5.3. The following `.java` file contains the exception handling stubs for `Panel_Type`:

```
package Cruise_Control_System_package;

class EHS_Panel_Type {

  EHS_Panel_Type() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void signal_engine_on_ssyncEH() {
    // FILL IN THE METHOD BODY IF NEEDED
  }

  void signal_engine_off_ssyncEH() {
    // FILL IN THE METHOD BODY IF NEEDED
  }

  void signal_accelerator_ssyncEH() {
    // FILL IN THE METHOD BODY IF NEEDED
  }

  void signal_brake_ssyncEH() {
    // FILL IN THE METHOD BODY IF NEEDED
  }

  void signal_on_ssyncEH() {
    // FILL IN THE METHOD BODY IF NEEDED
  }

  void signal_off_ssyncEH() {
    // FILL IN THE METHOD BODY IF NEEDED
  }

  void signal_resume_ssyncEH() {
    // FILL IN THE METHOD BODY IF NEEDED
  }

  void init_applet_archiEH() {
    // FILL IN THE METHOD BODY IF NEEDED
  }

  void start_applet_archiEH() {
    // FILL IN THE METHOD BODY IF NEEDED
  }

  void stop_applet_archiEH() {
    // FILL IN THE METHOD BODY IF NEEDED
  }

  void destroy_applet_archiEH() {
    // FILL IN THE METHOD BODY IF NEEDED
  }
}
```

We have a method for each semi-synchronous interaction of `Panel_Type`, together with a method for each architectural interaction of `P`. Note that each method of the first (resp. second) group has the same name as the corresponding interaction augmented with suffix `_ssyncEH` (resp. `_archiEH`).

The following `.java` file instead contains the internal action stubs for `Panel_Type`:

```
package Cruise_Control_System_package;

class IAS_Panel_Type {

  IAS_Panel_Type() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void update() {
    // FILL IN THE METHOD BODY
  }

  void beep() {
    // FILL IN THE METHOD BODY
  }
}
```

In this case, we have a method for each of the two internal actions of `Panel_Type`, which must necessarily be filled in by the software developer. ■

5.2.6. Translating `choice`

The choice operator expresses a selection among a certain number of alternative behaviors described through process terms. A choice-based process term is translated into a `switch-case` statement, whose condition is given by an invocation of the static method `choice()` defined in the class `ElemMeth` of package Sync.

There are two cases that must be addressed in order to translate the choice operator. The first one is the case where every process term involved in the choice starts with an action prefix operator. In this case, the method `choice()` is directly employed, which accepts as input an array of objects of class `ChAct`, each of which contains a boolean guard expressing the possible constraint under which the corresponding starting action is enabled (default value `true`). Should one of the starting actions be an interaction, an additional piece of information is contained in the corresponding object, which is a reference to the `Port` object associated with the interaction. Method `choice()` returns the index (within the array) of the starting action selected for execution.

A starting action is enabled (and hence can be selected for execution) if its guard evaluates to `true` and – in the case of a synchronous interaction – the corresponding `Port` object is ready to communicate. If all the starting actions with guard evaluating to `true` are synchronous interactions, method `choice()` waits – and the thread that contains it passivates – until one of the associated `Port` objects is ready to communicate. If all the guards of the starting actions evaluate to `false`, method `choice()` returns a negative value. Normally, at most one starting action is enabled, as the guards associated with alternative actions are usually in conflict with each other. Should this not be the case, i.e., if several starting actions are enabled, a probabilistic mechanism is applied to select one of those actions. This is useful when generating code from quantitative variants of PADL like Æmilia [5], in which the probability or the duration of actions can be expressed.

Based on the index returned by `choice()`, the `switch-case` statement invokes the method associated with the execution of the selected starting action. This method is `send()` or `receive()` in the case of an interaction, whereas for an internal action it is the corresponding method in the related IAS. The invocation of this method is followed in turn by the translation of the process term prefixed by the selected action. In the default clause, which comes into play when a negative value is returned by `choice()`, process term `stop` is invoked by assigning `null` both to `nextBehavEq` and to `actualPars`.

The second case is the one in which some of the process terms involved in the choice do not start with an action prefix operator. If one of these process terms is `stop`, then nothing has to be added for it in the `ChAct` array and the `switch-case` statement, because it is selected by default whenever the other involved process terms cannot be selected. If instead one of these process terms is a nested choice, then a flattening of the nested choice takes place during the translation.

Example 5.6. We complete the first omitted part of Ex. 5.3 by showing the definition of method `defineBehavEquations()` for the class associated with `Panel_Type`:

```
void defineBehavEquations() {

  Unallocated = new BehavioralEquationInterface() {
    public void behavEqCall() {
```

```
    _Unallocated();
  }
  private void _Unallocated() {
    try {
      init_applet.receive();
    } catch(UnattachedPortException e) {
      exception_Panel_Type.init_applet_archiEH();
    }
    try {
      start_applet.receive();
    } catch(UnattachedPortException e) {
      exception_Panel_Type.start_applet_archiEH();
    }
    nextBehavEq = Active;
    actualPars  = null;
  }
};

Active = new BehavioralEquationInterface() {
  public void behavEqCall() {
    _Active();
  }
  private void _Active() {
    switch (
      ElemMeth.choice(
        new ChAct[] {
          new ChAct(true, signal_engine_on),
          new ChAct(true, signal_accelerator),
          new ChAct(true, signal_brake),
          new ChAct(true, signal_on),
          new ChAct(true, signal_off),
          new ChAct(true, signal_resume),
          new ChAct(true, signal_engine_off),
          new ChAct(true, stop_applet)
        }
      )
    )
    {
      case 0:
        try {
          signal_engine_on.send();
        } catch(NotReadyPortException e) {
          exception_Panel_Type.signal_engine_on_ssyncEH();
        }
        nextBehavEq = Checking;
        actualPars  = new Object[] {signal_engine_on.success()};
        break;
      case 1:
        try {
          signal_accelerator.send();
        } catch(NotReadyPortException e) {
          exception_Panel_Type.signal_accelerator_ssyncEH();
        }
        nextBehavEq = Checking;
        actualPars  = new Object[] {signal_accelerator.success()};
        break;
      case 2:
        try {
          signal_brake.send();
        } catch(NotReadyPortException e) {
          exception_Panel_Type.signal_brake_ssyncEH();
        }
        nextBehavEq = Checking;
        actualPars  = new Object[] {signal_brake.success()};
        break;
      case 3:
        try {
          signal_on.send();
        } catch(NotReadyPortException e) {
          exception_Panel_Type.signal_on_ssyncEH();
        }
        nextBehavEq = Checking;
        actualPars  = new Object[] {signal_on.success()};
        break;
      case 4:
        try {
          signal_off.send();
        } catch(NotReadyPortException e) {
          exception_Panel_Type.signal_off_ssyncEH();
        }
        nextBehavEq = Checking;
        actualPars  = new Object[] {signal_off.success()};
        break;
      case 5:
        try {
          signal_resume.send();
        } catch(NotReadyPortException e) {
          exception_Panel_Type.signal_resume_ssyncEH();
        }
        nextBehavEq = Checking;
```

```
        actualPars  = new Object[] {signal_resume.success()};
        break;
      case 6:
        try {
          signal_engine_off.send();
        } catch(NotReadyPortException e) {
          exception_Panel_Type.signal_engine_off_ssyncEH();
        }
        nextBehavEq = Checking;
        actualPars  = new Object[] {signal_engine_off.success()};
        break;
      case 7:
        try {
          stop_applet.receive();
        } catch(UnattachedPortException e) {
          exception_Panel_Type.stop_applet_archiEH();
        }
        nextBehavEq = Inactive;
        actualPars  = null;
        break;
      default:
        nextBehavEq = null;
        actualPars  = null;
    }
  }
};

Checking = new BehavioralEquationInterface() {
  public void behavEqCall() {
    _Checking((Boolean)actualPars[0]);
  }
  private void _Checking(boolean success) {
    switch (
      ElemMeth.choice(
        new ChAct[] {
          new ChAct(success == true, null),
          new ChAct(success == false, null)
        }
      )
    )
    {
      case 0:
        internal_Panel_Type.update();
        nextBehavEq = Active;
        actualPars  = null;
        break;
      case 1:
        internal_Panel_Type.beep();
        nextBehavEq = Active;
        actualPars  = null;
        break;
      default:
        nextBehavEq = null;
        actualPars  = null;
    }
  }
};

Inactive = new BehavioralEquationInterface() {
  public void behavEqCall() {
    _Inactive();
  }
  private void _Inactive() {
    switch (
      ElemMeth.choice(
        new ChAct[] {
          new ChAct(true, start_applet),
          new ChAct(true, destroy_applet)
        }
      )
    )
    {
      case 0:
        try {
          start_applet.receive();
        } catch(UnattachedPortException e) {
          exception_Panel_Type.start_applet_archiEH();
        }
        nextBehavEq = Active;
        actualPars  = null;
        break;
      case 1:
        try {
          destroy_applet.receive();
        } catch(UnattachedPortException e) {
          exception_Panel_Type.destroy_applet_archiEH();
        }
        nextBehavEq = Unallocated;
        actualPars  = null;
        break;
```

```
          default:
            nextBehavEq = null;
            actualPars  = null;
        }
      }
    };
}
```

As can be noted, we have a `BehavioralEquationInterface` for each of the four behavioral equations of `Panel_Type`. ■

5.3. Preservation of Architectural Properties

We conclude by discussing the issue of guaranteeing that the properties proved at the architectural level – through the techniques illustrated in Sect. 4 – are preserved at the code level. Since we have taken an approach based on automatic code generation, property preservation should be achieved by construction. In other words, the translation from PADL to Java illustrated before should have been defined in a way that ensures property preservation. We now investigate this by separately considering the code generated for thread coordination, the code generated for translating behavioral equations, and the code provided for filling in stubs.

The code generated for coordinating threads cannot infringe the preservation of architectural properties, up to the methods for handling the exceptions that semi-synchronous or architectural interactions may raise. In fact, the code for thread coordination is completely generated in an automatic way by means of package `Sync`. As far as the system topology is concerned, this is built in the `RunnableArchi`-implementing class in the same way as prescribed by the second section of the PADL description. Moreover, both PADL and `Sync` adhere to the same communication model. On the PADL side, each interaction is given three qualifiers: input vs. output, synchronous vs. semi-synchronous vs. asynchronous, uni vs. and vs. or. Each interaction is then translated into an invocation of method `receive()` or `send()` defined in the corresponding `Port` object, depending on whether it is an input or an output interaction, respectively. Additionally, the kind of this `Port` object – synchronous vs. semi-synchronous vs. asynchronous, uni vs. and vs. or – is the same as that of the interaction.

Each behavioral equation occurring in an AET definition is translated into an inner state class of the corresponding `RunnableElem`-implementing class. The translation proceeds by induction on the syntactical structure of the process term occurring on the right-hand side of the behavioral equation. The way in which the translation is carried out, together with the way in which the thread execution flow proceeds according to the order established by the invocations of the behavioral equations, ensures the preservation of the AET behavior, up to the methods for translating internal actions.

As a consequence, the preservation of architectural properties critically depends on the way in which the software developer manually fills in EHSs and IASs. Here we shall consider only IASs, as EHSs can be treated similarly.

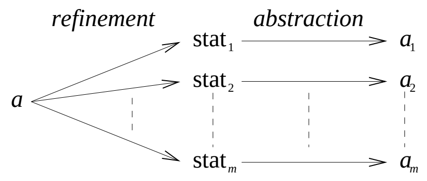


Figure 6: Refinement of an internal action and related statement abstraction

In order to be able to reason about architectural property preservation, we have to compare the internal actions and the corresponding sequences of Java statements on the same process algebraic ground. As shown in Fig. 6, each of the Java statements into which an internal action is refined during the translation process can be abstractly viewed as a fresh action. The following theorem provides a sufficient condition for ensuring the preservation of an architectural property. For the sake of simplicity, as mentioned at the beginning of Sect. 4, we concentrate on deadlock freedom. Below, we denote by $\simeq_{\rm B}$ the observational equivalence of [34], which is a refinement of $\approx_{\rm B}$ that turns out to be a congruence with respect to alternative composition too.

Theorem 5.7. Let T be the process algebraic description of the behavior of a thread and a be an internal action occurring in T. Let $a_1, a_2, \dots, a_m$ be the fresh actions abstracting the statements into which a is translated and T' be the process algebraic description of the behavior of the thread obtained from T by replacing every occurrence of $a\,.\,\_$ with $a_1\,.\,a_2\,.\,\dots\,.\,a_m\,.\,\_$. Let H be the set of internal actions occurring in T or T'. Whenever T is deadlock free and $a\,.\,\texttt{stop}/H \simeq_{\rm B} a_1\,.\,a_2\,.\,\dots\,.\,a_m\,.\,\texttt{stop}/H$, then T' is deadlock free as well.

Proof Since $\simeq_{\rm B}$ is a congruence with respect to all the process algebraic operators, from $a\,.\,\texttt{stop}/H \simeq_{\rm B} a_1\,.\,a_2\,.\,\dots\,.\,a_m\,.\,\texttt{stop}/H$ it follows that $T/H \simeq_{\rm B} T'/H$, hence $T/H \approx_{\rm B} T'/H$ because $\approx_{\rm B}$ is coarser than $\simeq_{\rm B}$. Since T is deadlock free, $\approx_{\rm B}$ preserves deadlock freedom, and deadlock freedom is expressed in terms of interactions (which cannot belong to H), it follows that T' is deadlock free as well. ■

Note that, in the theorem above, it is not necessarily the case that all of the actions $a_1, a_2, \ldots, a_m$ associated with the Java statements provided by the software developer belong to H. As an example, one of such actions may correspond to an invocation of `send()` or `receive()` or of a method such as `behavEqCall()` belonging to a state class. Fortunately, in practice both cases are prevented from occurring by the fact that the `Port` objects – which contain methods `send()` and `receive()` – and the `RunnableElem`-implementing class instances – which contain the state classes – are not visible within the stubs.

In order to preserve architectural properties, we provide some guidelines that the software developer should follow when filling in the stubs for handling exceptions and internal actions. The purpose of these guidelines is to achieve as much as possible the benefits of code generation, i.e., speeding up system implementation in a way that conforms by construction to the architectural model of the system:

- No synchronized or thread control methods – like `wait()` and `notify()` – should be invoked within the stubs, as their invocations would be internal to the threads but at the same time could affect the way threads communicate with each other.

- No further thread should be created within the stubs, as this would have an observable impact on the system topology and on thread coordination.

- There should be no variables/objects that are visible from several stub classes. This means that all the data shared by several threads should be exchanged only through suitable units of package `Sync`.

- The stub method associated with the first internal action following an invocation of `send()` or `receive()` should copy every object passed in that invocation, and all the stub methods associated with the subsequent internal actions should work on those copies of the objects. This would avoid interferences among threads stemming from the fact that `send()` always keeps a reference to the passed objects – so that it can be defined within `Sync` in a way that supports arbitrarily many parameters of arbitrary types – and such objects may be modified by the stub method associated with some internal action.

- All the exceptions that can be raised when executing a stub method should be caught – or prevented from being raised – inside the method, rather than propagating to the `RunnableElem`-implementing class.
- Nonterminating statements should not occur within stub methods.

We conclude by observing that we have found stubs more appropriate than abstract classes because the former allow the generated code to be compiled, including invocations of stub methods as they are concrete. In order to effectively remind the software developer to fill in the stubs, it would suffice to automatically introduce a statement in the definition of each stub method, which prints out an explicative message whenever an empty stub method is invoked at run time.

6. Conclusion

In this paper, we have extended a typical process algebraic architectural description language 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 the language without compromising its usability, we have shown that the architectural compatibility check and the architectural interoperability check can be easily adapted to cope with the presence of nonsynchronous interactions. Moreover, we have illustrated an architecture-inspired approach to the generation of multithreaded object-oriented code from process algebraic architectural descriptions containing an arbitrary combination of synchronous and nonsynchronous interactions in such a way that properties proved at the architectural level are preserved at the code level.

On the modeling and verification side, our work constitutes – as far as we know – the first systematic attempt to deal with semi-synchronous and asynchronous communications in process algebraic architectural description languages. Although we have focused on PADL (the interested reader is referred to [1] for a comparison with similar notations), we believe that our ideas can be applied to the other process algebraic architectural description languages appeared in the literature with minor modifications.

Concerning future work, we observe that, in the case of asynchronous interactions, the semantic model underlying a process algebraic architectural description may have infinitely many states due to the additional implicit components behaving like unbounded buffers. As shown in [13], it is hard to reason about component-based systems under an asynchronous semantics, because many properties such as deadlock freedom are undecidable. 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, which requires further investigation.

On the code generation side, there are several related work. First of all we mention ArchJava [2]. This is an extension of Java aiming at the unification of software architecture with implementation, in order to ensure that the implementation conforms to the architectural specification with respect to communication integrity. According to this property, each component in the implementation may only communicate directly with the components to which it is connected in the architecture.

Our approach differs from ArchJava in several ways. First, it does not extend Java, but generates Java code from process algebraic architectural descriptions. In our approach the developer is then required to fill in some stubs to complete the code for the behavior of the threads, thus giving a certain degree of flexibility. The price to be paid is that the guidelines may be violated, whereas a similar situation is not possible in ArchJava. Second, our approach adopts a richer communication model, implemented and transparently made available through package `Sync`. This guarantees a property even stronger than communication integrity: implementation threads directly communicate only with the threads they are connected to in the architectural description in the way prescribed by the architectural description itself with respect to communication synchronicity (synchronous, semi-synchronous, asynchronous) and communication multiplicity (uni, and, or). Third, since it keeps the architectural description language separated from the implementation language, our approach provides a higher-level support than ArchJava for the preservation of behavioral properties. On the other hand, the strong integration between architecture and implementation endows ArchJava with useful dynamic capabilities, like run-time creation of components – although communication integrity places restrictions on the way in which their instances can be used – and connections among them.

We then mention C2SADEL [33]. This is an architectural description language tied to the C2 style, which combines the usual architectural concepts with type theory. Type checking is used to analyze the architectural descriptions for consistency by unifying corresponding operations required and provided by different components. Moreover, Java code can be automatically generated from C2SADEL descriptions. Since type checking is a static analysis technique, while the architectural features on which we focus are behavioral and concerned with a rich communication model, we believe that our approach can guarantee the preservation of more complex properties than C2SADEL.

In the future, we plan to investigate the combination of our approach with those discussed before. In particular, we would like to investigate the applicability of our approach to C2SADEL, in order to take advantage of both type checking and behavioral analysis from the architectural level to the code level. Similarly, we would like to experiment our approach with ArchJava – by generating ArchJava code instead of Java code – in order to exploit the complementary strengths of the two approaches.

On the application side, we recall that the performance-oriented version Æmilia [5] of PADL is the input language of TwoTowers [7], an open-source software tool for the functional verification, security analysis, and performance evaluation of software architectures. TwoTowers is being extended in order to include the capability of expressing semi-synchronous and asynchronous interactions, as well as to investigate the absence of architectural mismatches through the modified compatibility and interoperability checks. Moreover, we have recently implemented another tool called PADL2Java, which translates PADL descriptions into multithreaded Java code according to the approach presented in this paper. PADL2Java will soon be integrated in TwoTowers, in order to construct an architecture-centric toolset encompassing modeling, verification, and implementation of software architectures. This will allow us to investigate the effectiveness and the scalability of our techniques when tackling larger case studies.

With regard to our toolset, we would like to integrate it with software model checking tools, like Java PathFinder [39], and to define specific rules for static analysis tools, like TPTP [28]. The reason is that the preservation at the code level of the properties proved at the architectural level is guaranteed only if (the underlying platform is correct and) the software developer follows the guidelines provided in Sect. 5.3 when filling in IAS and EHS stubs. Having a software model checker such as PathFinder – possibly driven by techniques like, e.g., those developed in [36, 26, 17] – available within TwoTowers would permit the verification of the overall system after the possible intervention of the software developer, whereas customized static analysis tools such as TPTP may be exploited for guiding the previously mentioned intervention.

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] J. Aldrich, C. Chambers, and D. Notkin, *"ArchJava: Connecting Software Architecture to Implementation"*, in Proc. of the *24th Int. Conf. on Software Engineering (ICSE 2002)*, IEEE-CS Press, pp. 187–197, Orlando (FL), 2002.

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

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

[5] S. Balsamo, M. Bernardo, and M. Simeoni, *"Performance Evaluation at the Software Architecture Level"*, in [12]:207–258.

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

[7] M. Bernardo, *"TwoTowers 5.1 User Manual"*, `http://www.sti.uniurb.it/bernardo/twotowers/`, 2006.

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

[9] M. Bernardo and E. Bontà, *"Preserving Architectural Properties in Multithreaded Code Generation"*, in Proc. of the *7th Int. Conf. on Coordination Models and Languages (COORDINATION 2005)*, LNCS 3454:188–203, Namur (Belgium), 2005.

[10] M. Bernardo and E. Bontà, *"Non-Synchronous Communications in Process Algebraic Architectural Description Languages"*, in Proc. of the *2nd European Conf. on Software Architecture (ECSA 2008)*, LNCS 5292:3–18, Paphos (Cyprus), 2008.

[11] 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.

[12] M. Bernardo and P. Inverardi (eds.), *"Formal Methods for Software Architectures"*, LNCS 2804, 2003.

[13] D. Brand and P. Zafiropulo, *"On Communicating Finite-State Machines"*, in Journal of the ACM 30:323–342, 1983.

[14] C. Canal, E. Pimentel, and J.M. Troya, *"Specification and Refinement of Dynamic Software Architectures"*, in Proc. of the *1st Working IFIP Conf. on Software Architecture (WICSA 1999)*, Kluwer, pp. 107–126, San Antonio (TX), 1999.

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

[16] 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.

[17] S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith, *"Modular Verification of Software Components in C"*, in IEEE Trans. on Software Engineering 30:388–402, 2004.

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

[19] R. Cleaveland and O. Sokolsky, *"Equivalence and Preorder Checking for Finite-State Systems"*, in [6]:391–424.

[20] K. Czarnecki and S. Helsen, *"Feature-Based Survey of Model Transformation Approaches"*, in IBM Systems Journal 45:621–645, 2006.

[21] X. Deng, M.B. Dwyer, J. Hatcliff, and M. Mizuno, *"Invariant-based Specification, Synthesis, and Verification of Synchronization in Concurrent Programs"*, in Proc. of the *24th Int. Conf. on Software Engineering (ICSE 2002)*, ACM Press, pp. 442–452, Orlando (FL), 2002.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley, 1995.

[23] D. Garlan, *"Formal Modeling and Analysis of Software Architectures: Components, Connectors, and Events"*, in [12]:1–24.

[24] D. Garlan, R. Allen, and J. Ockerbloom, *"Exploiting Style in Architectural Design Environment"*, in Proc. of the *2nd ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE 1994)*, ACM Press, pp. 175–188, New Orleans (LA), 1994.

[25] D. Gelernter, *"Generative Communication in Linda"*, in ACM Trans. on Programming Languages and Systems 7:80–112, 1985.

[26] D. Giannakopoulou, C.S. Pasareanu, and J.M. Cobleigh, *"Assume-Guarantee Verification of Source Code with Design-Level Assumptions"*, in Proc. of the *26th Int. Conf. on Software Engineering (ICSE 2004)*, IEEE-CS Press, pp. 211–220, Edinburgh (UK), 2004.

[27] R.J. van Glabbeek, *"The Linear Time - Branching Time Spectrum I"*, in [6]:3–99.

[28] S. Gütz and O. Marquez, *"TPTP Static Analysis Tutorial"*, `http://www.eclipse.org/tptp/`, 2006.

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

[30] 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.

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

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

[33] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor, *"A Language and Environment for Architecture-Based Software Development and Evolution"*, in Proc. of the *21st Int. Conf. on Software Engineering (ICSE 1999)*, IEEE-CS Press, pp. 44–53, Los Angeles (CA), 1999.

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

[35] 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.

[36] P. Parizek, F. Plasil, and J. Kofron, *"Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker"*, in Proc. of the *30th IEEE/NASA Software Engineering Workshop (SEW-30)*, IEEE-CS Press, pp. 133–141, Columbia (MD), 2007.

[37] A. Poggi and G. Rimassa, *"An Efficient and Flexible C++ Library for Concurrent Programming"*, in Software Practice and Experience 28:1437–1463, 1998.

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

[39] W. Visser, K. Havelund, G.P. Brat, S. Park, and F. Lerda, *"Model Checking Programs"*, in Automated Software Engineering Journal 10:203–232, 2003.