"ADORE", a Logical Meta–model Supporting Business Process Evolution

Sébastien Mosser[a,b], Mireille Blay–Fornarino[c]

[a]*INRIA Lille–Nord Europe, LIFL CNRS UMR 8022, Univ. of Lille 1, FRANCE*
[b]*SINTEF IKT, Oslo, NORWAY,* `sebastien.mosser@sintef.no`
[c]*I3S CNRS UMR 6070, Univ. of Nice, FRANCE,* `blay@polytech.unice.fr`

Abstract

The Service Oriented Architecture (SOA) paradigm supports the assembly of atomic services to create applications that implement complex business processes. Since "real–life" processes can be very complex, composition mechanisms inspired by the *Separation of Concerns* paradigm (*e.g.*, *features*, *aspects*) are good candidates to support the definition and the upcoming evolutions of large systems. We propose ADORE, *"an Activity meta–moDel supOrting oRchestration Evolution"* to address this issue. The ADORE meta-model allows process designers to express in the same formalism business processes and *fragment* of processes. Such *fragments* define additional activities that aim to be integrated into other processes and adequately support their evolution. The underlying logical foundations of ADORE allow the definition of interference detection rules as logical predicate, as well as the definition of consistency properties on ADORE models. Consequently, the ADORE framework supports process designers while they design and then apply evolutions on large processes, managing the detection of interferences among fragments and ensuring that the composed processes are consistent and do not depend on the order of the composition.

Keywords: SOA; Business Processes; Sep. of Concerns; Logical Composition.

1. Introduction

In the Service Oriented Architecture (SOA) paradigm an application is defined as an assembly of services that typically implements mission-critical business processes [1]. Services are loosely–coupled by definition, and complex services are built upon basics ones using composition mechanisms. These compositions describe how services will be orchestrated when implemented, and are created by business process specialists. In this context, a large part of the intrinsic complexity of an application is shifted into the definition of business processes. To perform such a task, process designers use "*composition of services*" mechanisms (*e.g.*, orchestrations of services) and realize their use cases as message exchanges between services. A factor that contributes to the difficulty of developing complex SOA applications is the constant evolution of the underlying business. As a consequence, a business expert initially designs its own process, which is then enriched by evolution added by other experts, *e.g.*, the *security* expert, the *persistence* expert, the *sales* expert.

This situation emphasizes the need for *separation of concerns* (SoC) mechanisms as a support to the design and the evolution of complex business processes. In the SoC paradigm, concerns are defined separately, and finally assembled into a final system using composition techniques. In the context of Soa, it leads us to compose orchestrations according to the upcoming evolutions, that is, to compose "*composition of services*". The complexity of building the final behavior that integrates all the concerns together is delegated to an algorithm.

The contribution of this paper is to support the evolution of business process through a compositional approach, *i.e.*, to automatically support at the model level the evolution of complex business processes through the composition of smaller artifacts. We propose Adore to fulfill this goal. Adore means *"an Activity meta–moDel supOrting oRchestration Evolution"* and supports a compositional approach for process evolution. Evolutions are reified as models describing smaller orchestrations of services to be composed to produce the evolved system.

2. Running Example: Exploring Pictures Folksonomies

To illustrate this work, we use the PicWeb system. This example was initially designed for a MSc lecture focused on *"Workflow & Business Processes"*, given in two universities (*i.e.*, Polytech'Nice school of engineering and University of Lille 1). As a consequence, it follows Soa methodological guidelines, positioning it as a typical usage of a Soa for experimental purposes [2]. PicWeb is a subpart of a larger legacy system called jSeduite[1], which deliver information to several devices in academic institutions. It supports information broadcasting from external partners (*e.g.*, transport network, school restaurant) to several devices (*e.g.*, smart-phone, PDA, desktop, public screen). jSeduite is currently deployed and daily used in three institutions (*i.e.*, Polytech'Nice, Ies Clément Ader and Irsam), where large plasma screens are used to broadcast public information. PicWeb is a subpartof jSeduite, retrieving a set of pictures annotated with a given *tag* from existing partner services. It implements a business process called `getPictures`. Based on a received *tag*, the process retrieves all the available pictures associated with *tag* from usual repositories (*e.g.*, flickr, provided by Yahoo!). In the remaining paragraphs, we assimilate PicWeb to this particular business process. We represent in Fig. 1(a) the first implementation of PicWeb, as initially defined in jSeduite. This process *(i)* receives a `tag` (a_0), *(ii)* asks a registry for the Api key associated to Flickr® (a_1), *(iii)* retrieves the relevant pictures from the repository (a_2) and finally *(iv)* replies this set to the user (a_3).

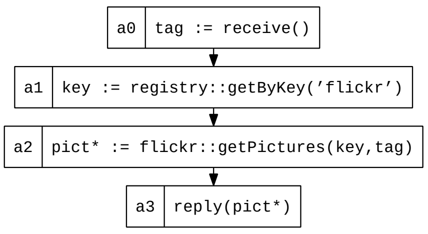
[1] `http://www.jseduite.org`

Need for evolutions in PicWeb. Soa systems are highly versatile, as they need to be continuously adapted to their underlying business. We illustrate this need in PicWeb through the change history of the system. These evolutions were driven by changes in the business needs, impacting the business process. For example, flickr conditions changed to restrict the number of times its service can be called per day. The business process had to evolve (*e.g.*, introducing a cache to reduce call frequency) to tackle this unforeseen change. Each evolution implies to change an activity graph increasingly complex, enforcing the need for SoC mechanisms. The final process represented in Fig. 1(b) (currently up and running) was built according to the five following situations. It was first enhanced to also address the picasa service (b_1, b_2), provided by Google. We secondly add a cache mechanism in front of the service calls (c_1, c_2, c_3). To limit the size of the retrieved set up to a given threshold, we thirdly introduce a truncation mechanism (d_1). Then, we introduce a shuffling activity (e_1) to obtain different pictures each time. Finally, we introduce a timer mechanism to log the execution time of the shuffle activity (f_1, f_2).

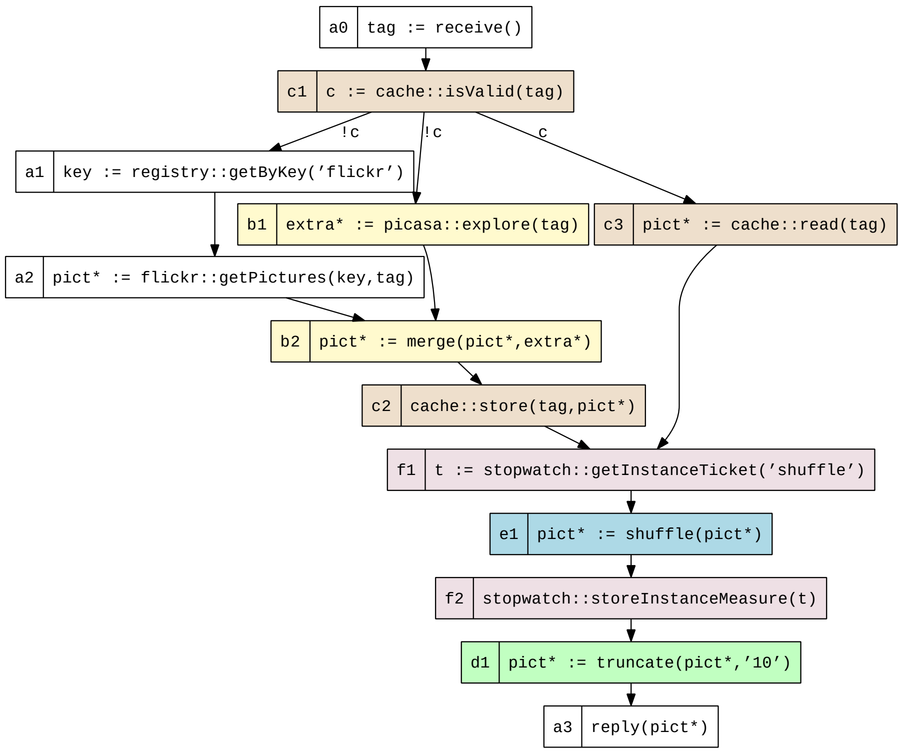
Challenges. As we saw, even on a simple example, the need for evolution to react to business changes in the context of business process is important. Therefore we consider in this article evolution as the integration of a set of changes in a legacy process. These changes are expressed separately by different stakeholders, and often introduce activities to be executed concurrently with the legacy process (*e.g.*, invoking picasa should not impact the invocation of flickr). Thus, the mechanism used to support these evolutions should not introduce unwanted waiting between activities. More critical interferences such as non–determinism can also be introduced in legacy processes. The manual detection of such interferences is difficult in front of thousands of activities and relations. Moreover, allowing a human to edit manually a business process does not ensure strong properties (*e.g.*, preservation of the preexisting behavior) on the obtained process. The key idea of this contribution is to provide a formal meta–model supporting such evolution in an automatic way, *i.e.*, composing unforeseen changes with legacy processes and supporting interferences detection.

3. ADORE: Modeling Business Process with First–Order Logic

We define Adore to address these challenges. The Adore activity meta–model is directly inspired by the Bpel language grammar expressiveness. The Bpel is defined as *"a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners"* [3]. The Bpel defines nine different kinds of atomic activities (*e.g.*, service invocation, message reception and response) and seven composite activities (*e.g.*, sequence, flow, if/then/else), plus additional mechanisms such as transactions and message persistence. We decide to extract from the Bpel all concepts necessary *"to describe the behavior of a business process"*. From the activity expressiveness point of view, Adore can be seen as a simplification of the Bpel concepts. We focus here on the definition of an activity kernel that support


a0
tag := receive()
a1
key := registry::getByKey('flickr')
a2
pict* := flickr::getPictures(key,tag)
a3
reply(pict*)

(a) Initial version of PicWeb (v_0)


a0
tag := receive()
c1
c := cache::isValid(tag)
!c
!c
c
a1
key := registry::getByKey('flickr')
b1
extra* := picasa::explore(tag)
c3
pict* := cache::read(tag)
a2
pict* := flickr::getPictures(key,tag)
b2
pict* := merge(pict*,extra*)
c2
cache::store(tag,pict*)
f1
t := stopwatch::getInstanceTicket('shuffle')
e1
pict* := shuffle(pict*)
f2
stopwatch::storeInstanceMeasure(t)
d1
pict* := truncate(pict*,'10')
a3
reply(pict*)

(b) Final version of PicWeb (v_5)

As Adore *is inspired by graph theory, we use a graph-based representation of business processes. Vertexes represent activities (*e.g.*, message reception, service invocation), and edges represent causality relations. A* `a` $\to$ `b` *relation means that* `b` *will wait for the end of* `a` *to start its own execution. A* `a` $\xrightarrow{v}$ `b` *relation strengthens the previous semantics, and conditions the start of* `b` *to the value of* `v`*. In this example, relations are combined using a conjunctive semantics (except for f_1, as its incomming branches are syntactically exclusive). Activities a_1 and b_1 are concurrently executed when condition c in c_1 is evaluated to false. Colors annotate activities added during the same evolution.*

Figure 1: Graph-based representation of PicWeb business processes: from v_0 to v_5

the definition of business processes in the large, without introducing a strong dependency with BPEL–specific concepts. Thus, we do not consider in ADORE such notions (*e.g.*, correlation set). A coarse–grained description of the ADORE meta–model is depicted in FIG. 2 using the class–diagram formalism.

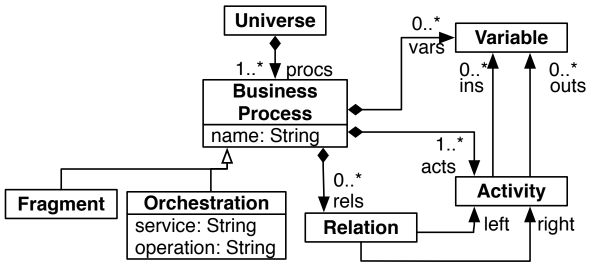


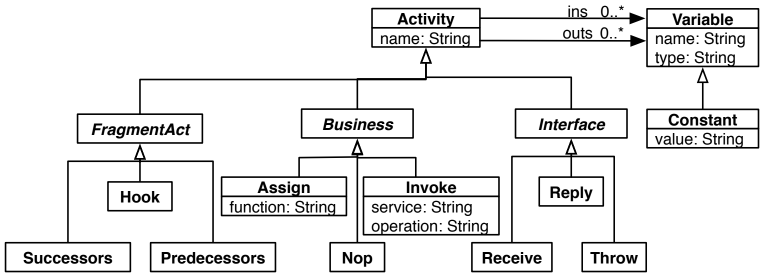
Figure 2: Global overview of the ADORE meta–model

In ADORE, a business process is defined as a partially ordered set of activities. It defines several variables, used as input or outputs by its activities. Binary relations are used to schedule the activity set. A process can model an orchestration (*i.e.*, an executable business process) or a fragment (*i.e.*, a partial business process to be integrated into an orchestration to implement an evolution). The root of the meta–model is reified as the universe, that is, the element that contains all the available processes.

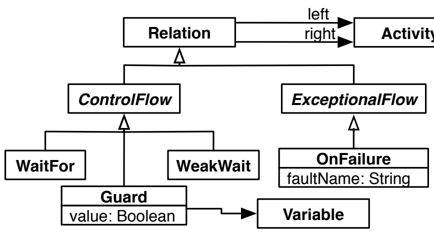
The `Activity` concept represents an elementary task realized by a given business process. An activity uses a set of inputs variables (`ins`), and assigns its result in a set of outputs variables (`outs`). The different types of activities that can be defined in ADORE (see FIG. 3(a)) include *(i)* service invocation (`Invoke`), *(ii)* variable assignment (`Assign`) *(iii)* fault reporting (`Throw`), *(iv)* message reception (`Receive`), *(v)* response sending (`Reply`), and *(vi)* the null activity (`Nop`), which is used for synchronization purpose. Consequently, from an expressiveness point of view, the ADORE meta-model contains all the atomic activities defined in the BPEL normative document except the `wait` (stopwatch activity) and the `rethrow` that implement syntactic sugar[2].

ADORE defines four different types of relationships between activities (see FIG. 3(b)) to reify BPEL control–structures. All relations are expressed using binary operators. ADORE can define simple wait between activities through the `WaitFor` operator (related to the `sequence` structure in the BPEL, and denoted as $a \prec a'$). Conditional branches in a control–flow are represented using `Guard` relations (related to the `if` structure in the BPEL, and denoted as $a \overset{v}{\prec} a'$). A BPEL `flow` is modeled in ADORE as the absence of relation. As the complete description of the ADORE meta–model [4] is out of the scope of this paper, we

[2]*E.g.*, in ADORE, a `rethrow` can be performed by *catching* a fault with an `OnFailure` relation and then use a `Throw` activity to re–throw the fault.



(a) `Activity` class hierarchy



(b) `Relation` class hierarchy

Figure 3: Extracts of the ADORE meta–model

will focus here on `WaitFor` and `Guards`, which are expressive enough to cover the example studied in this paper. Nevertheless, to support the expressiveness of the BPEL, we also define the `WeakWait` relation, used to reify predecessors' disjunction [5] (*i.e.*, non–determinism in the order of events). Finally, a fault catching mechanism is defined through the `OnFailure` relations.

Contrarily to the BPEL, the intrinsic purpose of ADORE is to support the evolution of orchestrations. To perform such a goal, we extend the previously defined meta–model to represent *partial* processes (see FIG. 3(a)). Such `Fragment` are not suitable for direct execution: they aim to be integrated into existing `Orchestrations`. As a consequence, they hold three kinds of *special* activities, dedicated to this goal: *(i)* a `Hook` which represents the point where the fragment will be integrated, *(ii)* a `Predecessors` to represent hook' predecessors in the targeted orchestration and finally *(iii)* a `Successors` to represent hook' successors. To illustrate this point, we depict in FIG. 4(a) a fragment of process used to *truncate* a data set $Data^\star$. We decide to represent fragment specific activities using a *dashed* notation instead of a plain one, to make their intrinsic nature explicit[3]. This fragment introduces a d_1 activity between its *hook* point (h) and the associated predecessors (circled P). The $Data^\star$ variable is actually a *ghost* variable, related to the existing behavior (as it is used by the *hook*). We illustrate in FIG. 4(b) the expressiveness of ADORE. In this fragment, we

[3]They refer to the behavior of the process where the fragment will be integrated to.

concurrently introduce a call to the PICASA service (b_1), which retrieves relevant pictures using the Tag ghost variable. Then, we introduce an activity (b_2) in the preexisting control–flow ($\mathbb{P} \rightarrow h \rightarrow \mathbb{S}$) to merge the pictures retrieved from this new source with the ones obtained through the legacy invocation (*i.e.*, the $Pict^{\star}$ ghost variable).

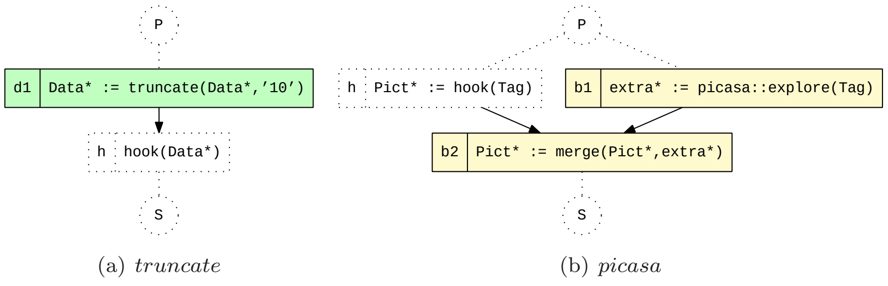

P
d1
Data* := truncate(Data*,'10')
h
hook(Data*)
S
(a) *truncate*
P
h
Pict* := hook(Tag)
b1
extra* := picasa::explore(Tag)
b2
Pict* := merge(Pict*,extra*)
S
(b) *picasa*

Figure 4: Example of process fragments

Links with Many–sorted First–order Logic. ADORE is formalized through the definition of each concept using the many–sorted first–order logical formalism. The goal of this work is to build a solid basis used to support evolution algorithm definition and interference detection. Moreover, the usage of first–order logic definitions allows us to easily map this formal model into an executable implementation, using the PROLOG language[4]. Instead of providing an extensive description of the logical foundations of ADORE, we will introduce the associated concepts on the fly. We use the following conventions in the next definitions: *(i)* we denote with a * symbol a set (*e.g.*, $2 \in \mathbb{N}$, $\{1, 2\} \in \mathbb{N}^{\star}$) and *(ii)* we access to the content of a tuple using eponymous functions (*e.g.*, the *name* function gives access to the *name* part of the tuple). A more detailed version of the formal model is available in Appendix A.

Consistency Properties. An ADORE model (*i.e.*, a model conforms to the ADORE meta–model) assumes the respect of several consistency rules[5]. These properties reify constraints expressed over the meta–model and consequently restrict the ADORE expressiveness into entities compatible with the evolution mechanism described in the upcoming sections. For example, processes are isolated in ADORE, and activities and variables are uniquely contained by one process. A formal description of the consistency rules is available in Appendix B. We describe here only the three properties used as strong assumptions by the evolution algorithm described in SEC. 4.2. In the following sections, we define a *path*

[4]The ADORE implementation (started in 2007) is available on the project website (`www.adore-design.org`), defined as 8,000 lines of PROLOG and 3,000 line of Java wrapper code.

[5]The implemented engine throws an exception when it identifies a constraint violation.

between two activities a and a' (denoted as $a \rightarrow a'$) as the transitive closure of the available relations in the process to be followed to reach a' from a.

Property 1. ***(fragment coherence)****. Let $f \in \mathcal{F}$ a fragment. This fragment defines at least one path which connects* predecessors*,* hook *and* successors*.*

$$\forall f \in \mathcal{F},\ \mathbb{P}(f) \rightarrow hook(f) \wedge hook(f) \rightarrow \mathbb{S}(f)$$

Property 2. ***(process isolation)****. Let $p \in \mathcal{P}$ a process. Activities and variables contained in p are locally scoped and as a consequence cannot be shared with other processes. The only way to break this isolation is to use the evolution process to accurately merge several elements into a new one.*

$$\forall p \in \mathcal{P}, \forall a \in acts(p) \Rightarrow \nexists p' \in \mathcal{P},\ p' \neq p \wedge a \in acts(p')$$
$$\forall p \in \mathcal{P}, \forall v \in vars(p) \Rightarrow \nexists p' \in \mathcal{P},\ p' \neq p \wedge v \in vars(p')$$

Property 3. ***("entry–to–exit" path existence)****. Let $p \in \mathcal{P}$. All exit points of p are connected to its entry point through a path.*

$$\forall p \in \mathcal{P}, \forall a \in acts(p), a = exit(p) \Rightarrow entry(p) \rightarrow a$$

Execution Semantics. The execution semantics is based on boolean logic, as we associate to each activity a_i a boolean formula computed over the set of its predecessors, denoted as $\varphi(a_i)$. An in–depth description of the φ computation can be found in [6]. The key idea of this execution semantics is to mimic the semantics of the Bpel language, completing the expressiveness compatibility between Adore and Bpel. A more detailed version of the formal model is available in Appendix C. We describe here example of trigger formulas computed for the PicWeb process described in Fig. 1(b).

$$\begin{array}{rcl@{\qquad}rcl}
\varphi(c_1) & = & end(a_0) & \varphi(a_1) & = & end(c_1) \wedge \neg c \\
\varphi(c_3) & = & end(c_1) \wedge c & \varphi(b_2) & = & end(a_2) \wedge end(b_1) \\
\varphi(f_1) & = & end(c_2) \vee end(c_3) & & & (c_2 \text{ and } c_3 \text{ are exclusive})
\end{array}$$

4. Stepwise Evolution of Business Processes

Using Adore, we support the evolution of business processes through the weaving of fragments into existing orchestrations. The goal of this section is *(i)* to describe the mechanisms used to perform a *weave* and *(ii)* to formally describe the different properties ensured by the algorithm. A preliminary version of the described algorithm was initially described in [7].

4.1. Action–based weaving

As the Adore meta–model aims to compose business processes together, we need to formally define mechanisms to interact with Adore models. State–of–art approaches such as the Mof reflexive Api [8] use an action language to perform such a goal. The PRAXIS approach advocates an action–based approach of model representation: *"Every model can be expressed as a sequence of elementary construction operations. The sequence of operations that produces a model is composed of the operations performed to define each model element" [9].* It restricts the set of operations defined by the MOF to four, allowing one to *(i)* `create` a model element, *(ii)* `delete` a model element, *(iii)* set a property in a model element (`setProperty`) and finally *(iv)* set a reference from a model element to another one (`setReference`).

Considering ADORE as a meta–model dedicated to business processes, we decide to specialize the concepts expressed in PRAXIS to build a *domain–specific* language dedicated to the handling of ADORE elements. In this context, we define two elementary actions (`add` & `del`), and two *composite* actions defined as *macros* over the elementary ones (`replace` & `discharge`). We decide to use a *functional* representation of the actions, considering that the execution of an action returns a **new** universe instead of modifying the existing one through a side–effect. To lighten the description of the mechanisms, we refer to ADORE *elements* in the same way one refers to *"model elements"* when talking about class instances. We denote as *Action* the set of all existing actions. To lighten the description of the logical formulas, we indifferently accept for an *element* the usage of the element itself or its identifier (its *name*).

Execution. An action is performed on a given universe $u \in \mathcal{U}$. We denote as $do : Action \times \mathcal{U} \rightarrow \mathcal{U}$ the function used to concretely execute an action. Such execution cannot fail and always returns a new universe. Considering an ordered list[6] of actions $A = (\alpha_1, \ldots, \alpha_n) \in Actions^{\star}_{<}$, we define the execution of A on u as the *functional folding* [11] of do (denoted as do^+).

$$
\begin{array}{rrcl}
do^+ : & Action^{\star}_{<} \times \mathcal{U} & \rightarrow & \mathcal{U} \\
 & (A, u) & \mapsto & \left\{ \begin{array}{lcl} A = \emptyset & \Rightarrow & u \\ A \neq \emptyset & \Rightarrow & do^+(tail(A), do(head(A), u)) \end{array} \right.
\end{array}
$$

Atomic operations: ***`Add`*** *&* ***`Del`**.* Using these actions, one can change a *container* element (*e.g.*, a business process or an universe) by adding (respectively removing) an element inside the container. To lighten the description, we assume that the *container* exists in the universe used to perform the action. Executing an elementary action only modifies the *container* element, *ceteris paribus.* We use a *kind* index to differentiate the different kinds of elements handled by the (*e.g.*, add_r represents the add of a relation). These actions are idempotent.

- `Add`: This operation is used to create an element and insert it into another one, *e.g.*, adding an activity into a business process. It tooks as input a ground term (the kind of concept to be added, *e.g.*, *activity*), the *element*

[6]We consider a *list* as usually defined in the literature [10], with function such as *head* to retrieve its first element, *tail* to access to the others and $\overset{+}{\leftarrow}$ to append an element at its end. We use the $\mathcal{E}^{\star}_{<}$ notation to denote a "list of $\mathcal{E}$"

to be added and its future *container*. We use a term of arity two to represent it: $add_{kind}(element, container)$.

- `Del`: The semantics of a *del* action is to remove an element in the original model, and the execution of such an action returns an universe which is contained by the original one. The deletion of a non–existing element is accepted, and basically returns the original universe. We use a term of arity two to represent it: $del_{kind}(element, container)$.

$$\begin{array}{rcll} do(add_k(e,c),u) = u' & \Rightarrow & do(add_k(e,c), do(add_k(e,c),u)) = u' & \text{(Idemp.)} \\ do(del_k(e,c),u) = u' & \Rightarrow & do(del_k(e,c), do(del_k(e,c),u) = u' & \text{(Idemp.)} \\ u & = & do(del_k(e,c), do(add_k(e,c),u)) & (del \circ add) \end{array}$$

Composite actions: ***`Replace`*** *& **`Discharge`**.* Since the previously described actions are elementary and technically–oriented, we define more complex actions by combining them. We define these actions as macros, that is, actions which can be refined into elementary ones. The execution semantics associated to these macro–actions is equivalent to the execution of the sequence of elementary actions associated to a macro, statically computed before starting the execution.

We use *logical substitutions* as defined by Stickel [12] to allow one to build new elements by modifying existing ones. According to this definition, a substitution component is *"an ordered pair of a variable v and a term t written as $v \leftarrow t$. A substitution component denotes the assignment of the term to the variable or the replacement of the variable by the term."*. Thus, a substitution is *"a set of* substitution components *with distinct first elements, that is, distinct variables being substituted for. Applying a substitution to an expression results in the replacement of those variables of the expression included among the first elements of the substitution components by the corresponding terms. The substitution components are applied to the expression in parallel, and no variable occurrence in the second element of a substitution component is replaced even if the variable occurs as the first element in another substitution component"*. The application of substitution θ to expression A is denoted by $A\theta$. The composition of substitutions $\theta\sigma$ denotes the substitution whose effect is the same as first applying substitution θ, then applying substitution σ; that is, $A(\theta\sigma) = (A\theta)\sigma$ for every expression A.

- `Replace`: The *replace* action allows one to change all usages of an element (named *old*) by another one (named *new*). The *replace* action is modeled as a term of arity four: the *kind* of replacement, the *old* element, the *new* one, and the container p. We define a short notation $r(old, new, p)$ which makes implicit the kind of replacement to lighten the descriptions. Executing a *replace* action means to add the *new* element, *substitute* all usage of *old* in p by a usage of *new*, and then delete the *old* element. In ADORE, we only allow the replacement of variables and activities, contained by a given process $p \in \mathcal{P}$. A replacement is valid only between elements from the same kind (*i.e.*, an activity $a \in \mathcal{A}$ cannot be replaced by a variable $v \in \mathcal{V}$). For example, to replace a variable v by a variable v' in a given process p, the execution of $r(v, v', p)$ actually *(i)* adds v' in p, propagates the use of v' in *(ii)* the activities that used v ($prop_a$) and *(iii)* the `Guard` relations that used v ($prop_r$), and finally *(iv)* deletes v.

$$
\begin{aligned}
&r(a, a', p) \in \mathit{Action},\ a \in \mathcal{A},\ a' \in \mathcal{A},\ p \in \mathcal{P} : \qquad\qquad \theta = \{a \leftarrow a'\}\\
&\quad clean_{aft} = \{del_r(a \prec x, p) \mid \exists a \prec x \in rels(p)\}\\
&\quad clean_{bef} = \{del_r(x \prec a, p) \mid \exists x \prec a \in rels(p)\}\\
&\quad rpl = (del_a(a, p),\ add_a(a', p))\\
&\quad reorder_{aft} = \{add_r(a' \prec x, p) \mid \exists a \prec x \in rels(p)\}\\
&\quad reorder_{bef} = \{add_r(x \prec a', p) \mid \exists x \prec a \in rels(p)\}\\
&\leadsto\ () \overset{+}{\hookleftarrow} clean_{aft} \overset{+}{\hookleftarrow} clean_{bef} \overset{+}{\hookleftarrow} rpl \overset{+}{\hookleftarrow} reorder_{aft} \overset{+}{\hookleftarrow} reorder_{bef}\\
&r(v, v', p) \in \mathit{Action},\ v \in \mathcal{V},\ v' \in \mathcal{V},\ p \in \mathcal{P} : \qquad\qquad \theta = \{v \leftarrow v'\}\\
&\quad prop_a = \{r(a, a\theta, p) \mid \exists a \in acts(p), v \in vars(a)\}\\
&\quad prop_r = \{(del_r(r, p), add_r(r\theta, p)) \mid \exists r = a \overset{v}{\prec} a' \in rels(p)\}\\
&\leadsto\ (add_v(v', p)) \overset{+}{\hookleftarrow} prop_a \overset{+}{\hookleftarrow} prop_r \overset{+}{\hookleftarrow} del_v(v, p)
\end{aligned}
$$

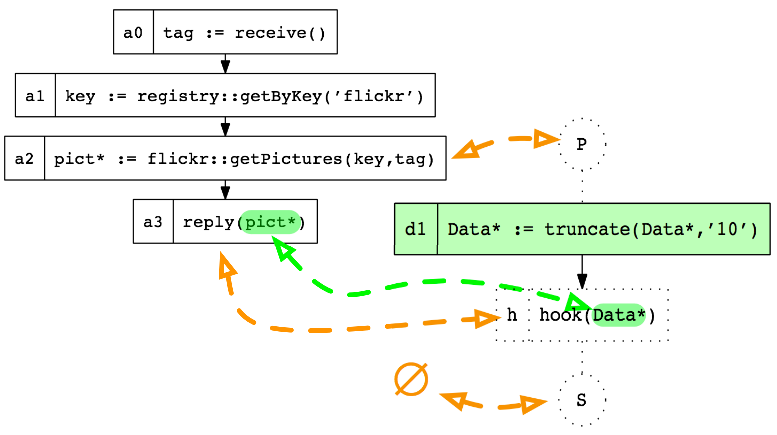
- `Discharge`: This composite action allows one to *discharge* the contents of a given process $p \in \mathcal{P}$ into another one (called p'). As a consequence, the discharged process is destroyed and its contents is added inside the targeted one. For example, this action is used to add the whole content of a fragment inside its targeted orchestration. We model this action as a term of arity two: $d(p, p')$.

$$
\begin{aligned}
&d(p, p') \in \mathit{Action},\ p \in \mathcal{P},\ p' \in \mathcal{P} :\\
&\quad del = \{del_r(r, p) \mid \exists r \in rels(p)\}\\
&\qquad\qquad \cup \{del_a(a, p) \mid \exists a \in acts(p)\} \cup \{del_v(v, p) \mid \exists v \in vars(p)\}\\
&\quad add = \{add_v(v, p') \mid \exists v \in vars(p)\} \cup \{add_a(a, p') \mid \exists a \in acts(p)\}\\
&\qquad\qquad \cup \{add_r(r, p') \mid \exists r \in rels(p)\}\\
&\leadsto\ () \overset{+}{\hookleftarrow} del \overset{+}{\hookleftarrow} (del_p(p), add_p((p', \emptyset, \emptyset, \emptyset)) \overset{+}{\hookleftarrow} add
\end{aligned}
$$

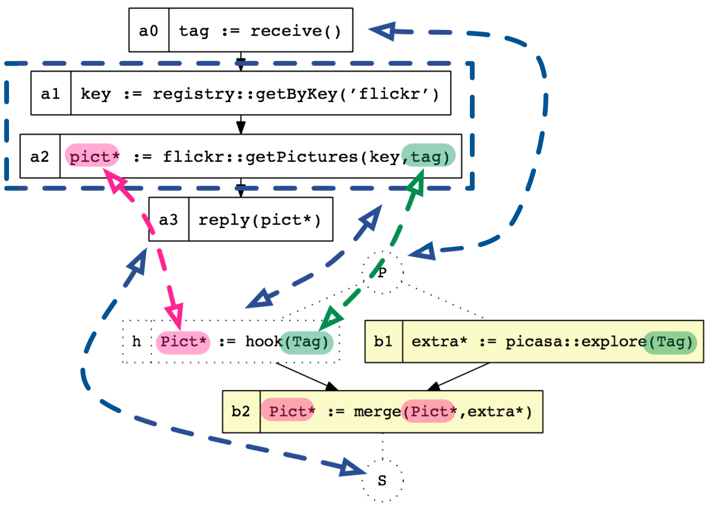
4.2. The SimpleWeave *Algorithm*

Using the SimpleWeave algorithm, a process designer can implement an evolution through the integration of a fragment $f \in \mathcal{F}$ into another process (*i.e.*, orchestration or fragment) $p \in \mathcal{P}$. For example, one can asks Adore to integrate the *truncate* fragment (depicted in Fig. 4(a)) into the initial implementation of PicWeb, obtaining an enhanced version (see Fig. 5(a)). We depict in Fig. 5(b) how the *picasa* fragment (see Fig. 4(b)) is composed with the initial process to implement such an evolution.

To perform the SimpleWeave, the designer must express *where* the fragment will be woven (*e.g.*, in the previous case, the a_3 activity). This target is defined as a set of activities ($B \in \mathcal{A}^\star$, called a *block*). We also use a binding function β to let the user bind the ghost variables with concrete ones at weave time (*e.g.*, in the previous case, the function associates the $Data^\star$ ghost element


a0
tag := receive()
a1
key := registry::getByKey('flickr')
a2
pict* := flickr::getPictures(key,tag)
a3
reply(pict*)
P
d1
Data* := truncate(Data*,'10')
h
hook(Data*)
S

(a) Truncating the picture set


a0
tag := receive()
a1
key := registry::getByKey('flickr')
a2
pict* := flickr::getPictures(key,tag)
a3
reply(pict*)
P
h
Pict* := hook(Tag)
b1
extra* := picasa::explore(Tag)
b2
Pict* := merge(Pict*,extra*)
S

(b) Invoking PICASA

Figure 5: Business process evolutions

to the concrete *pict** variable) . We modeled this weaving *directive* as a term ω of arity three: $\omega(f, B, \beta) \in \mathit{Directive}$, where $f \in \mathcal{F}$ the fragment to be integrated, $B \in \mathcal{A}^\star$ its target and $(\beta : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{B})$ the binding function. In the previous example, the two following directive are used: *(i)* $\omega(\mathit{truncate}, \{a_3\}, \equiv)$ to introduce the *truncate* fragment, and *(ii)* $\omega(\mathit{picasa}, \{a_1, a_2\}, \equiv)$ to introduce the *picasa* fragment.

Both uses the variable type equivalence function ($\equiv$) to bind ghost variables to concrete ones (this is the default). Thus, concrete variables to be used will be identified based on their structural equivalence with the ghosts ones:

(i) Data$^\star \equiv$ *pict** as they are both typed as "set of pictures", and *(ii)* $Tag \equiv tag$ as they are both typed as strings. Using a user–given function allows ADORE to let the user explicitly *choose* the variable to be used in front of multiple candidates (*e.g.*, if multiple strings were defined in the process). Nevertheless, one of our validation (see SEC. 6) case study defined 139 directives, and the type equivalence function was used for 135 directives.

The SIMPLEWEAVE algorithm takes as input a directive $\omega(f, B, \beta)$, and a process p ($B \subset acts(p)$). The algorithm starts by discharging the content of f in p. As a consequence, all elements (variables, activities and relations) defined in the fragment are now part of the targeted process, including ghost elements (*i.e.*, hook variables and fragment–specific activities). These ghosts must be eliminated from the process. The algorithm relies on two principles to perform this task: *(i)* it concretises the ghost activities of f (*i.e.*, *predecessors*, *successors* and *hook* are binded with existing elements) and *(ii)* it replaces the ghost variables of f by real ones according to the binding function β.

(i) Ghost Activities Vanishment. The ghost activities of the woven fragment must be replaced by the associated elements defined in p, according to B. Considering a directive denoted as $\omega(f, \{a\}, \beta)$, we denote as P_a the set of a predecessors, and S_a the set of a successors in p. The *hook* activity defined in f is denoted as h and its predecessors as P_h (respectively its successors as S_h). The *predecessor* ghost activity is denoted as $\mathbb{P}$ (respectively $\mathbb{S}$ for the *successors* ghost activity). These elements are formally defined as the following (the instantiation on the previous *truncate* example is given on the right part):

$$
\begin{array}{rcll}
\text{Let } p & \in & \mathcal{P},\ \omega(f, \{a\}, \beta) \ \in \ Directive,\ a \in acts(p), & \\
P_a & = & \{\ \alpha \mid \exists \alpha < a \in rels(p)\} \in \mathcal{A}^\star & \mid \quad = \ \{a_2\} \\
S_a & = & \{\ \alpha \mid \exists a < \alpha \in rels(p) \wedge a \rightarrow \alpha\} \in \mathcal{A}^\star & \mid \quad = \ \emptyset \\
\mathbb{P} & = & \mathbb{P}(f),\ \mathbb{S} \ = \ \mathbb{S}(f),\ h \ = \ hook(f) & \\
P_h & = & \{\ \alpha \mid \exists \alpha < h \in rels(f)\} \in \mathcal{A}^\star & \mid \quad = \ \{d_1\} \\
S_h & = & \{\ \alpha \mid \exists h < \alpha \in rels(f)\} \in \mathcal{A}^\star\} & \mid \quad = \ \emptyset
\end{array}
$$

The ghost activities vanishment follows the three following principles:

- P_1: **Linking the untargeted behavior with the new one**. Each relation existing in the original process between a predecessor of a ($\pi \in P_a$) and a must be propagated[7] to add such relations when a relation is defined between $\mathbb{P}$ and a newly added activity ($\phi \in acts(f)$). It also propagates the relation existing between a and successors of a ($\sigma \in S_a$), following the same principle. The actions generated to perform this task ensure that the legacy predecessors and successors are well–connected with the newly added activities.

- P_2: **Linking the new behavior with the concrete hook**. Each relation existing between the hook predecessors ($\pi \in P_h$) and the *hook* itself

[7]**Propagate**: "the spreading of something into new regions" (`wordnet.princeton.edu`).

(respectively the *hook* and a successors of the hook — $\sigma \in S_h$) must be propagated to take care of the newly added activities. The actions generated for this task ensure that the activities added through the fragment are well connected with the targeted activity.

- P_3: **Deleting Ghost Activities**. All relations between $\mathbb{P}$, $\mathbb{S}$ and h must be deleted. After that these ghost activities can be deleted too. As a consequence, after performing these actions, the ghost elements defined in f are definitively destroyed.

This step generates the following actions in the example described in FIG. 5:

$$\begin{array}{lll} P_1 & : & \{add_r(a_2 \prec d_1, v_0)\}, \qquad P_2 \ : \ \{add_r(d_1 \prec a_3, v_0)\} \\ P_3 & : & \{del_r(\mathbb{P} \prec d_1, v_0), del_r(d_1 \prec h, v_0), del_r(h \prec \mathbb{S}, v_0), \\ & & \quad del_a(\mathbb{P}, v_0), del_a(h, v_0), del_a(\mathbb{S}, v_0)\} \end{array}$$

(ii) Ghost Variable Replacement. The ghost variables used in the fragment must be replaced by the concrete ones, preexisting in the legacy behavior. We use the binding function β to perform such a task[8]. For each input ghost variable (respectively output), the β function is invoked to identify a matching candidate existing in the real inputs (respectively outputs) of the targeted activity. Assuming that there is a single candidate matched, the algorithm generates the associated *replace* directives, and picks up the next ghost variable. When applied to the previous example, this step generates the replacement of $data^\star$ by $pict^\star$ (*i.e.*, $r(data^\star, pict^\star, v_0)$).

The binding process can also identify two other situations: *(i)* there is no candidate found for this ghost variable or on the contrary *(ii)* there are multiple candidates. At the implementation level, an error is raised in front of such a situation. Recent work with the requirement engineering community allows us to also consider alternative strategies [2].

Handling Blocks of Activities. The previous principles were defined over a *single* concrete activity. Considering the target defined in a directive as an activity block $B = \{a_1, \ldots, a_n\} \in \mathcal{A}^\star$, we can generalize the two previously described principles to accurately tackle it. The key idea of this generalization is to compute the *firsts(B)* (respectively *lasts(B)*) elements of the activity blocks according to the partial order, and then compute the associated actions based on their predecessors (*preds(B)*) and successors (*succs(B)*). For example, in the example depicted in FIG. 5(b) uses a block $b = \{a_1, a_2\}$ as target. Thus, the predecessors of this block are assimilated to the predecessors of its first element, *i.e.*, $preds(b) = \{a_0\}$. In case of concurrent first elements (*e.g.*,, consider a block $b' = \{a_1, a_2, b_1\}$ in FIG. 1(b), $firsts(b') = \{a_1, b_1\}$), predecessors are computed as the union of the predecessor of each first element (in the previous example, this particular case returns the singleton $\{c_1\}$). The semantics associated to the successors is equivalent, using the *lasts* element of the block instead of the *firsts* ones.

[8]We assume that β is injective and deterministic.

Complete Algorithm. We describe in ALG. 1 the SIMPLEWEAVE algorithm. The algorithm starts by initializing local variables and generating the first action: discharging the fragment content into the targeted process (line 2). Then, it performs the ghost activity vanishment step (lines 5 $\to$ 10). The variable replacement step (lines 12,13) is performed before the return of the final action list (line 15). It is important to notice that actions are generated on the legacy processes (*i.e.*, the right part of the intensional definitions relies on the preexisting elements) and then cannot interfere with each others. We represents in FIG. 6 the result of the execution of SIMPLEWEAVE on the running example. As fragments are copied by the engine before the composition, it is possible to reuse the same fragment in multiple composition directives.

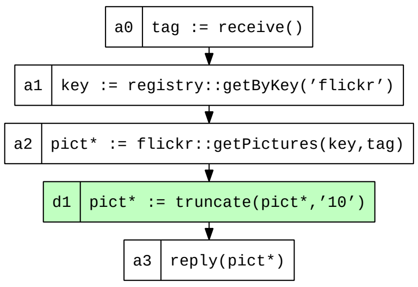
Algorithm 1 SIMPLEWEAVE: $\omega \times p \mapsto actions$

Require: $\omega = \omega(f, B, \beta) \in Directive,\ p \in \mathcal{P}$
Ensure: $r \in Actions^{\star}_{<}$

```
 1:                 $\{$ Initialization $\}$
 2: $h \hookleftarrow hook(f),\ \mathbb{P} \hookleftarrow \mathbb{P}(f),\ \mathbb{S} \hookleftarrow \mathbb{S}(f),\ r \hookleftarrow (d(f,p))$
 3: $F_B \hookleftarrow firsts(B),\ L_B \hookleftarrow lasts(B),\ P_B \hookleftarrow preds(B),\ S_B \hookleftarrow succs(B)$
 4:                 $\{$ (i) Ghost Activity Vanishment $\}$
 5: $r \overset{+}{\hookleftarrow} \{add_r((\pi,\alpha,l),p) \mid \exists \pi \in P_B,\ \exists a \in F_B,\ \exists(\pi,a,l) \in rels(p),\ \exists \mathbb{P} < \alpha \in rels(f),\ \alpha \neq h\}$   $\{P_1\}$
 6: $r \overset{+}{\hookleftarrow} \{add_r((\alpha,\sigma,l),p) \mid \exists a \in L_B,\ \exists \sigma \in S_B,\ \exists(a,\sigma,l) \in rels(p),\ \exists \alpha < \mathbb{S} \in rels(f),\ \alpha \neq h\}$   $\{P_1\}$
 7: $r \overset{+}{\hookleftarrow} \{add_r((a,\alpha,l),p) \mid \exists a \in acts(f),\ a \neq \mathbb{P},\ \exists(a,h,l) \in rels(f),\ \exists \alpha \in F_B\}$   $\{P_2\}$
 8: $r \overset{+}{\hookleftarrow} \{add_r((\alpha,a,l),p) \mid \exists a \in acts(f),\ a \neq h,\ \exists(h,a,l) \in rels(f),\ \exists \alpha \in L_B\}$   $\{P_2\}$
 9: $r \overset{+}{\hookleftarrow} \{del_r((\mathbb{P},a,l),p) \mid \exists(\mathbb{P},a,l) \in rels(f)\} \cup \{del_r((a,\mathbb{S},l),p) \mid \exists(a,\mathbb{S},l) \in rels(f)\}$   $\{P_3\}$
10: $r \overset{+}{\hookleftarrow} (del_a(\mathbb{P},p),\ del_a(h,p),\ del_a(\mathbb{S},p))$   $\{P_3\}$
11:                 $\{$ (ii) Ghost Variable Replacement $\}$
12: $r \overset{+}{\hookleftarrow} \{r(v,real,p) \mid \exists v \in inputs(h), \exists! real \in inputs(B),\ \beta(v,real)\}$   $\{Inputs\}$
13: $r \overset{+}{\hookleftarrow} \{r(v,real,p) \mid \exists v \in outputs(h), \exists! real \in outputs(B),\ \beta(v,real)\}$   $\{Outputs\}$
14: return $r$
```

4.3. From SIMPLEWEAVE *to* WEAVE

The previously described algorithm only allows the weaving of a single fragment into a single process. We define the final WEAVE algorithm as a folding of the SIMPLEWEAVE one. It takes as input a set of directives Ω, and the targeted process. The computed action list is returned to the caller as the concatenation of all the action list generated by the SIMPLEWEAVE algorithm. the immediate advantage of the WEAVE algorithm is its order–independence, as it considers is input directives as a set. Thus, executing WEAVE$(\{\omega_1, \omega_2\}, p)$ means to introduce the evolution modelled by ω_1 and ω_2 without ordering them. It is not possible with usual approach (*e.g.*, ASPECTJ) that relies on sequential weaving $(f \bullet g(x) = f(g(x)))$: with such systems, ω_1 will be woven on p, and the result



a0
tag := receive()
a1
key := registry::getByKey('flickr')
a2
pict* := flickr::getPictures(key,tag)
d1
pict* := truncate(pict*,'10')
a3
reply(pict*)

Figure 6: Process evolution: $do^{+}(\textsc{SimpleWeave}(\omega(\mathit{truncate}, \{a_3\}, \equiv), v_0), v_0)$

of this weaving will be used as target to weave ω_2. Thus, the weaving of ω_2 may captures elements introduced by the weaving of ω_1. ADORE prevents such a behavior through the computation of each action set (one for each composition directive) independently of the others. Sequential weaving is immediately supported by the approach, as the execution of algorithm output is endogenous and produces a new process (evolved): $\textsc{Weave}(\{\omega_2\}, \textsc{Weave}(\{\omega_1\}, p))$

Algorithm 2 $\textsc{Weave}$: $\Omega \times p \mapsto \mathit{actions}$

```
Require: $\Omega \in \mathit{Directive}^{\star},\ p \in \mathcal{P}$
Ensure: result $\in \mathit{Actions}^{\star}_{<}$
1: return $() \overset{+}{\hookleftarrow} \{\textsc{SimpleWeave}(\omega(f, B, \beta), p) \mid \exists \omega(f, B, \beta) \in \Omega\}$
```

5. Detecting Interferences During the Evolution Process

Interactions between the different evolutions may happen and then generate *interference*. This vocabulary is borrowed to the Aspect–oriented community: *"Aspects that in isolation behave correctly, may interact when being combined. When interaction changes an aspect's behaviour or disables an aspect, we call this interference"* [13]. Based on the meta–model previously defined and its associated logical representation, it is possible to verify a given process against declarative specifications of invariants. In this paper, we focus on non business interferences detection and show how the proposed algorithms ensure some properties, such as determinism. However, this mechanism can also be used to implement business invariants (*e.g.*, *"the shuffle activity must be monitored"*), specific to a given process. For the sake of concision, we only describe here invariants that are common to all business processes, and provided to the designers by the ADORE framework.

5.1. Interferences Detection: Concurrency

ADORE defines several rules to detect behavioral interferences, that is, unexpected behavior generated by interferences between the woven evolutions. We focus here on interferences driven by concurrency issues: *(i)* concurrent access to a variable and *(ii)* concurrent response sending.

Activity Exclusiveness. The *concurrent* interference described here intuitively relies on the exclusivity of activities execution. Informally, two activities a and a' are defined as *exclusive* (denoted as $a \otimes a'$) when they cannot be executed in parallel. We base the definition of this property on the boolean formula associated to each activity through ADORE execution semantics (see Appendix C). The property is computed as a comparison of all execution formulas defined in the execution path to identify the exclusion[9].

Concurrent Access to a Variable. We consider here the following situation: in addition to the previously described *truncate* evolution (applied on $\{a_3\}$), another user asks ADORE to introduce a *shuffle* evolution (applied on $\{a_2\}$), as described in FIG. 7. This evolution generates a concurrent access to the variable $pict^{\star}$, as both d_1 and e_1 use it as output.

Usual aspect ordering techniques cannot handle this situation, since the two different fragments are woven at different locations in the base process and then cannot be ordered using an ASPECTJ–like mechanisms[10]. Moreover, the fact that ADORE identifies such an issue is a strength to support the assessment of large processes after evolution. At the implementation level, this situation is automatically detected through the satisfaction of a logical rule. Such an event is raised to the user, who can identify its source and fix the problem (*e.g.*, by adding a *waitFor* relation between the two activities, in the previous case $e_1 \prec d_1$).

$$\begin{aligned}&\text{Let } p \in \mathcal{P},\ \exists a \in acts(p),\ \exists v \in outputs(a),\ \exists a' \in acts(p),\\ &\quad a' \neq a, v \in vars(a'),\ \neg(a \otimes a')\end{aligned}$$

Concurrent Termination. Considering two different evolutions that enhance a process with new exit activities, it is possible to obtain as a result of the evolution process a non–deterministic process, which can answer two different responses for the same input message. This situation can be identified by ADORE as the satisfaction of a logical rule. Identifying such a situation supports designers to identify unexpected (and potentially complex) conditions set in their processes.

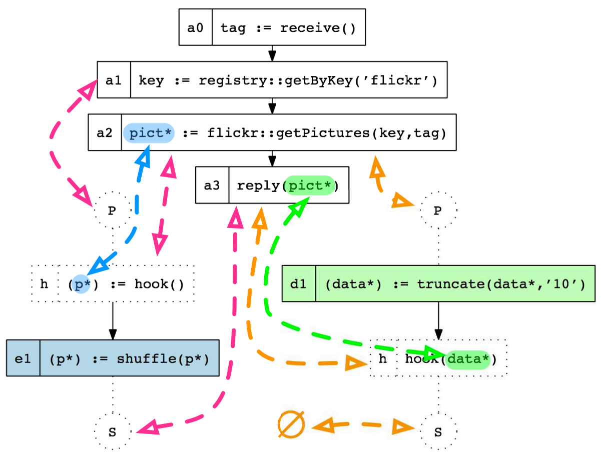
$$\text{Let } p \in \mathcal{P},\ \exists a \in exit(p), \exists a' \in exit(p), a' \neq a, \neg(a \otimes a')$$

5.2. Bad–smells: Equivalent Activities & Forgotten Directives

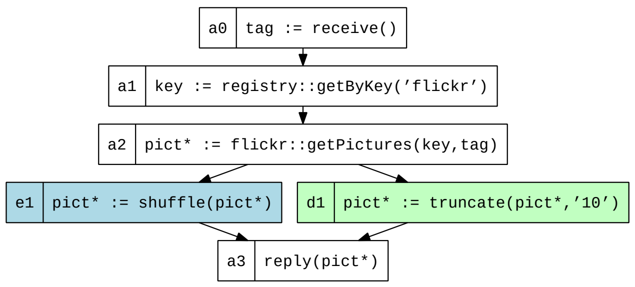
The previously described interferences were *dangerous* in terms of process execution. We focus now on the detection of *bad–smells* in the composition process. We identified two situations were such bad–smells can occur: *(i)* a fragment introduces an equivalent activity in a process and *(ii)* an activity equivalent to one used as a fragment target exists in the process.

[9]We consider as exclusive two formula φ and φ' since they hold exclusive conditions, or different events for the same activity (*i.e.*, *end* and *failure*).

[10]In this case, one has to rewrite the aspects to fix the inconsistency.


a0
tag := receive()
a1
key := registry::getByKey('flickr')
a2
pict* := flickr::getPictures(key,tag)
a3
reply(pict*)
P
h
(p*) := hook()
e1
(p*) := shuffle(p*)
S
P
d1
(data*) := truncate(data*,'10')
h
hook(data*)
S
∅

(a) Conflicting weaving: *shuffle* on a_2, *truncate* on a_3


a0
tag := receive()
a1
key := registry::getByKey('flickr')
a2
pict* := flickr::getPictures(key,tag)
e1
pict* := shuffle(pict*)
d1
pict* := truncate(pict*,'10')
a3
reply(pict*)

(b) Obtained (non–deterministic) process

Figure 7: Concurrent access interference: $pict^{\star}$ on $\{d_1, e_1\}$

Introducing Equivalent Activities. Multiple fragments can introduce service invocations that are equivalent by weaving the same fragment on several activities or by requesting the same service in different fragments. A logical predicate is used to identify the situation and adequately inform the designer.

$$\text{Let } \omega(f,B,\beta) \in \mathit{Directive},\ p \in \mathcal{P}, B \subset \mathit{acts}(p),\\ \exists \alpha \in \mathit{acts}(f), \exists \alpha' \in \mathit{acts}(p), \alpha \equiv \alpha'$$

Forgotten Weave. When a fragment f is woven on an activity a' that is equivalent to another activity a that is not yet woven with f, the designer may have forgotten to weave f on the activity a (*e.g.*, an access control policy). Such a situation is identified by ADORE and a suggestion is displayed to the designer.

$$\text{Let } \omega(f,B,\beta) \in \mathit{Directive},\ p \in \mathcal{P}, B \subset \mathit{acts}(p),\ \exists B' \subset \mathit{acts}(p) \backslash B,\\ B \equiv B',\ \omega(f,B',\beta') \notin \mathit{Directive}$$

5.3. Properties Ensured

The Weave algorithm supports designers when they need to introduce evolution in their processes, by automating the integration of evolutions (implemented as fragments). This integration ensures a set of properties, which helps designers to assess the composed process. When using the Weave algorithm to enrich a preexisting process, designers know that these properties are ensured *by the algorithm*, and can then focus on their business preoccupations.

Weave Property 1. *Directive Order Independence. For a given* Weave *execution, the way designers express the directives does not matter in the obtained result.* The Weave algorithm works on a set of directive Ω, which is by definition unordered. For each directive $\omega(f, B, \beta) \in \Omega$, the algorithm generates a set of actions, but never executes it. Consequently, weave directives are handled *in isolation*, and cannot interfere between each others.

Weave Property 2. *Determinism. The* Weave *algorithm is deterministic and ensures to always return the same result for a given set of weave directives.* The actions generated for each weave directive add new activities and relations in the legacy process, and only delete the ghost elements. The way relations are added is deterministic. According to the idempotency property of the elementary actions and the order–independence, the execution of the action set always produces the same process as a result.

Weave Property 3. *Path Preservation. An execution of the* Weave *algorithm may extend preexisting execution paths, or add new ones, but cannot destroy existing ones.* Let f a fragment. According to the *fragment coherence* property (Prop. 1), at least one path exists between $\mathbb{P}(f)$, $hook(f)$ and $\mathbb{S}(f)$. Let B the block used as weave target, P_B the preexisting predecessors of B and S_B its preexisting successors. By construction, the $\mathbb{P}(f)$ activity is mapped with all activities defined in P_B, and $\mathbb{S}(f)$ is mapped with all activities defined in S_B. The algorithm never deletes a preexisting relation. Consequently, existing paths are transitively conserved in the composed process.

Weave Property 4. *Execution Order Preservation. If the handled fragment does not bypass their hooks, the preexisting execution order is preserved by construction. In the other situations, such a bypass is considered as expected at the application level since it is defined in the fragment by the designer.* The path preservation property ensures that the preexisting execution flow is conserved, but does not ensure that the preexisting execution semantics is preserved. The only way of breaking the preexisting semantic is to weave a fragment that defines a path which does not contains its hook activity. Such a situation implies that the concrete successors of the targeted block of activity may then be executed even if the block was not executed.

Weave Property 5. *Condition Preservation. Using* Adore*, guards added by fragments and presexisting guards cannot interfere between each others and are preserved by construction, since the designer does not use a ghost variable in* *a guard.* The previous property ensures that the execution order is preserved. Even if we assume the respect of this property in this paragraph, it does not mean that preexisting conditions are always preserved. According to the process isolation property (Prop. 2), each variable defined in a process is unique, and contained by only one process. Variables interact between each others through the *ghost variable replacement* step of the Weave algorithm, where *hooked* variables are mapped to concrete ones. As a consequence, the only way for a fragment to interact with a preexisting guard is to define a new guard based on a *hooked* variable. A restriction of Adore expressiveness (a guard relation should not use as left part a ghost variable) ensures that the preexisting conditions are always preserved, since the execution order is preserved.

Weave Property 6. *Variable Initialization. Assuming that preexisting elements are initialized, the variables handled in a composed process are initialized too, by construction.* For a given fragment f, ghost variables are the only one which do not need to be initialized: they are replaced with concrete elements by the algorithm. Assuming that the handled processes (both fragment and targeted process) always initialize a variable before using it, the algorithm ensures that no uninitialized variable can be encountered in a composed process (thanks to the path preservation property P_{w_3}). At the verification of variable initializations relies on an exploration of all possible execution paths, which is a very time–consuming operation. As a consequence, it is faster to check this property on small artifacts, and ensure by construction that the composed (eventually huge) process respects this property.

Weave Property 7. *Process Response. For each received message, a response message will be generated by the composed process.* The *entry–to–exit* property (Prop. 3) ensures that a fragment f always defines a path between its entry point $\mathbb{P}(f)$ and an exit point ($\mathbb{S}(f)$, a *throw* or a *reply* activity). This property must also be true in the targeted process. The algorithm binds $\mathbb{P}(f)$ and $\mathbb{S}(f)$ to concrete activity. As a consequence, the newly added activities are connected to the process entry point (through $\mathbb{P}(f)$ binding) and to the preexisting exit point through $\mathbb{S}(f)$ binding (if needed). Consequently, we can ensure that each activity (which is not an exit point) is connected to at least one exit point in the composed process.

6. Validation

In addition to the design of PicWeb with Adore, we also illustrate the usage of the approach with two large case studies, published in separated contributions.

Cccms. The *Car Crash Crisis Management System* (Cccms) is realized as an answer for a common case study on Aom, according to a requirements document co–published by the University of McGill and the University of Luxembourg [14]. We use Adore to generate two systems in this context: *(i)* the business–only system (including only the business extensions, *i.e.*, evolution to be introduced in the initial process) and *(iii)* the final system (including both business extensions and non–functional concerns). Even if this case study is not driven by evolution needs, it allows us to validate ADORE according to the three following dimensions: *(i)* expressiveness, *(ii)* scalability and *(iii)* implementation. The complete details of this case study are available in [15].

- Expressiveness: The requirements document (29 pages) was defined by external stakeholders, which emphasizes the fact that ADORE can be used in real–life context. The special issue associated to this document evaluated the different approaches *w.r.t. "how well they describe the application of the approach to the* CCCMS *case study, the appropriate numbers of models and on the quality of the analysis of advantages and disadvantages of the AOM approach"*. The ADORE approach allows us to cover all the use cases defined in the document, without any methodological issue. The ADORE implementation of the CCCMS is composed by 8 orchestrations and 30 fragments. Moreover, interference detection mechanisms identify lacks in the requirements, at the process level (*e.g.*, data that were defined in a particular business process but never reused in the others).

- Scalability: The cognitive load indicator aims to quantify the intrinsic complexity of a business process for a human designer. It is defined as an coarse grained approximation of the Control Flow Complexity indicator [16]. For five business processes, the cognitive load of the final process was significantly higher than the sum of the cognitive loads of the basic element used to obtain it (*e.g.*, $247 > 6$). For the others, the two values were on the same scale. We also notice that up to 80% of a final business process is composed by activities that were not defined in its initial description.

- Implementation: The CCCMS case study was intrinsically defined as a very large case study. The final system (obtained after the execution of the composition directives) represents 1216 activities scheduled by 895 relations[11]. We summarize in TAB. 1 the number of times each algorithm is invoked and the number of actions performed on the initial model to execute the composition. The composition engine naively implemented to support the ADORE approach (plain translation of logical rules into Prolog predicates, without specific optimizations), consumes on an average computer[12] 68 seconds to build the final system (*i.e.*, main success scenario, business extensions and non-functional concerns). Using a *profiler*, we identified that the test of model invariants consumes 90% of the execution time. Even if such a check is by essence a non–polynomial problem, simple

[11]All the models (inputs artifacts and composed ones) are available on the case study web page: `http://www.adore-design.org/doku/examples/cccms/`

[12]MacBook Pro (2007), Mac OSX Tiger, Intel Core 2 Duo 2.4GHz, 2Go SDRAM 1066MHz

optimizations (*e.g.*, introducing *cuts* in the invariants to stop backtracking) can be done to reduce the time. However, the complete synthesis of such a large model in the scale of a minute is completely acceptable *w.r.t.* concurrent approaches.

Modeled System	WEAVE	$\|Action^\star\|$	Exec. Time
Business–driven CCCMS	34	2422	$\sim$ 5s
Including NF concerns	139	10838	$\sim$ 68s

Table 1: Algorithms usage (& associated actions) when modeling the CCCMS

JSEDUITE. The JSEDUITE application is based on a legacy implementation, daily used inside several academic institutions. It is used to broadcast information to user's devices in the context of academic institutions. The first version of this open–source system was released in 2005, and the current version represents 70,000 lines of code. We identified several issues while developing the system, as repetitive, time–consuming and error–prone situations. We perform a reverse engineering of the JSEDUITE system, using ADORE as a compositional approach to support its evolution[13]. This case study allows us to validate ADORE according to the two following dimensions: *(i)* the capture of a real–life evolution process at the business process level and *(ii)* the capitalization of such evolutions in a software product line.

- Evolution Capture: we based this study on the evolution made to the up and running systems deployed and daily used in three institutions. School users express needs *w.r.t.* the running systems, and business processes were impacted to tackle this unforeseen changes. The most important changes expressed by the users were the introduction of new sources of information in the system. Thus, the concurrency property provided by ADORE (as independent sources are called concurrently) was critical in this context to invoke up to 18 sources of information without introducing unwanted waits. The different fragments used to model each evolution simply define that the new source is called concurrently with the legacy process. The interference detection engine is then used to identify problematic situations and fix them.

- Capitalization: As each evolution applied to the system is implemented as a fragment, it is possible to obtain a version of JSEDUITE that does not contain it. It simply means not to weave this fragment on the legacy process. Users also express *policies* to be applied on sources of information, like shuffling or truncating the returned information. Thanks to the automated support provided by ADORE, we start to model a software product

[13]`http://www.adore-design.org/doku/examples/faros/start`

line of jSeduite that accurately capitalize all the evolutions done in the system since the beginning [17]. An ongoing 2–years project[14] funded by the French national agency of research pursue the validation of Adore in this context.

7. Related Work

Relations with Aspect–oriented approaches. According to the Ercim working group on software evolution [18], *aspect–oriented* approaches rely at a syntactic level on four elementary notions: *(i) joinpoints*, *(ii)*, *pointcuts (iii)*, *advice* and finally *(iv) aspects*. *Joinpoints* represents the set of well-defined places in the program where additional behavior can be added. In the context of Adore, we use activities to reify this notion. *Pointcuts* are usually defined as a set of joinpoints. In Adore, one can identify sets of activities as pointcuts using explicit declarations (*e.g.*, use $\{act_3, act_4\}$ activities as pointcuts) or computed declarations (*e.g.*, all activities calling the service *srv*). *Advice* describes the additional business logic to be added in the initial system. Adore represents systems as a set of business processes. We reify *advices* in an endogenous way as business processes called *fragment*. Finally, *aspects* are defined as a set of pointcuts and advices. Adore uses *composition directives* to bind fragments to set of activities.

The main originality of Adore *w.r.t.* the aspect–oriented approaches is its order–independence weaving, where on the contrary aspects are sequentially woven. Dedicated Aop platform must be used to support aspect concurrency [19], but their weaving is still sequential. One can found in the literature several research works which implement Aop mechanisms for Soa. For example, the Ao4Bpel framework [20] proposes a modification of a Bpel execution engine to support dynamic weaving of aspects. In [21], authors propose to use aspect at different levels (semantic analysis, Bpel engine & Bpel processes) to support the introduction of unforeseen features. These approaches handle Aop at the level of the Bpel engine. However, these approaches are syntactically driven and does not support properly the definition of reasoning mechanisms (*e.g.*, interference detection) or the insurance of weaving properties. On the contrary, Adore proposes a way to make operational such mechanisms.

Interference detection. State–of–the–art research work such as Mata [22] defines interference detection rules, based on critical pair analysis [23]. These mechanisms support designers by analyzing the given weaving directives, and then identify *(i)* dependencies (a directive produces an element consumed by another one) and *(ii)* conflicts (a directive avoids the application of another one). This notion supports the development of rule–based system, identifying conflicting situations such as "the rule r will delete an element matched by the rule r'" or "the rule r generates a structure which is prohibited according to

[14] ANR YourCast, call Emergence 2011, 221k€, PI: Pr. Blay–Fornarino.

the existing preconditions". We reuse these mechanisms on the set of woven directives to avoid weaving cycles for example. However, the detection mechanism proposed in this work only focus on directives, where ADORE supports the definition of any rules using plain logic. We provide with ADORE a set of detection rules that ensures the consistency of the system, complementing the ones provided by MATA. Nevertheless, ADORE also supports home–made rules provided by the user that implements business-driven invariants.

Ciraci *et al* [24] proposes Grace, a graph–based approach to support aspect interference detection, at the model level. Based on graphs that represent the structure of a given model, they define a set of graph–transformation that implements an operational semantics for these graphs. They extends a formal model–checker to automatically identify violations of invariants in the given models. As stated in SEC. 5, one can use ADORE to express business–specific invariants to be checked in the evolved models. ADORE is then complementary to Grace, focusing on behavior where their approach focuses on structure. Moreover, we recently combine ADORE with a tool that support structure composition [25], and as a consequence it should be possible to combine ADORE with Grace to also support graph–based invariant for the structure of the services involved in an SOA. However, the business process models deal with more coarse–grained artifacts than the object model used by Ciraci, and a gap have to be filled between the service–orientation and the object–representation. The graph–based approach was also extended to software architecture [26]. Instead of expressing changes in terms of architecture rewriting, we place ourselves at the level of business process and use ADORE fragments to capture changes. Weaving is used to apply these changes according to different evolution scenarios. We have not studied the cost of changes, and qualifying the choice of fragments using scoring (based on graph analysis) is one of our perspectives. ADORE and Grace are already positioned in an tool–chain *w.r.t.* their Aspect–oriented modelling roots [27].

Business Process & Workflow Evolution. The problem of the evolution of business processes is an old problem when considered in the large [28]. A particular attention is given here to the propagation of the evolution on the running instances of a given process [29], as the execution of such a process can last for several years. Weber *et al.* [30] propose a conversational case-based reasoning to support the evolution of a process class through the analysis of the changes made on the associated instances. Inheritance was also investigated to support business process evolution [31]. According to this approach, one can perform an evolution by *"sub–classing"* a given process. Thus, migrating a process instance from an old process definition to a new one is reduced as a class conformity problem. ADORE does not address the instance level, and only focus on the modelling of a process to be enacted in a dedicated workflow engine. Thus, the evolutions applied to a given ADORE process are not propagated to the running instances. We recently introduced a runtime dimension in ADORE based on complex–event processing to use events as the trigger of process adaptation [32]. The propagation of an evolution at the instance level is still an ongoing work.

Safe Transformations of Models. Introducing a set of evolutions in a legacy process can be seen as a model transformation. In the particular case of ADORE, we consider in this paper model–to–model transformation, where orchestrations and fragments are the input models, and the WEAVING algorithm the transformation to be applied. Contrarily to approaches that focuses on the modeling of transformations (*e.g.*, Stratego [33], ATL [34]), we only provide with ADORE a dedicated way to compose models in order to support their evolution. The main originality of ADORE *w.r.t.* this topic is its ability to apply transformation in isolation (see weaving properties 1 & 2), thanks to the underlying action–based approach. Generalizing the approach used in ADORE to support the definition of any composition algorithm is an ongoing work.

Heidenreich *et al.* [35] propose a component–based approach to combine model–transformation. According to Aßman [36], a system that supports composition of model transformation must be composed by *(i)* a model to represent the application of transformations, *(ii)* a language to express their composition and finally *(iii)* a technique to make operational the previous element. In ADORE, we use *directive* to model the application of transformations, set–theory to express their composition and finally an action–based approach to implement the transformations at runtime. Considering that several evolution algorithms can co–exist in the same domain (the complete ADORE framework define three additional evolution algorithms [4]), this kind of techniques will be necessary to rule the transformation system. This topic is currently explored as a part of the generalization mentioned in the previous paragraph.

Formal Service Composition. Diapason [37] is a formal framework used to verify, deploy and execute composition of services. Based on π–calculus, the approach is code–oriented: designers add *"evolution point"* in their orchestrations to support the upcoming modifications, at this particular point. ADORE is more flexible on this point, as we allow a fragment to be woven at any place in an orchestration. In Diapason, different execution paths (*i.e.*, traces) are computed thanks to simulation techniques. These traces are then analysed against properties defined using the logic-Diapason language. ADORE also supports the declarative specification of model invariant, and ensure that the evolved models are compliant *w.r.t.* these invariants. We provide default invariants (*i.e.*, deterministic termination), and the user can use the same formalism to express invariants dedicated to a given business domain. However, ADORE suffers from using the graph–based models as input for the invariant checker engine. Similar work in the UML domain [9] shows that trace–based constraint checking is more efficient than graph–based one in this domain. A perspective of this work is to investigate this point, simulating execution of ADORE process to obtain traces and then using these traces as input for the invariant checker.

Sequence Diagrams Weaving. In [38], the authors propose a method dedicated to support the weaving of aspects using sequence diagrams formalism. Behavioral aspects are defined as sequence diagrams too. The approach is integrated into a multi–view modeling approach (RAM, [39]). Aspects pointcuts are defined as events captured by the aspect and expressed in the base sequence diagram. The key idea of the approach is to support the matching of message sequences using a *partOf* semantic: an pointcut capturing the sequence of messages $m_1 \rightarrow m_2$ will match a more longer sequence such as $m_1 \rightarrow \mu \rightarrow m_2$. This mechanisms supports the introduction of multiple aspects on the same point, as the events added by an aspect will not interfere with the others. According to its weaving mechanisms [40], the approach supports the usage of multiple events as target, where usual Aop techniques capture a single one. From Klein's method, we keep three key points: *(i)* the usage of the same formalism to represent both aspect and base program, *(ii)* the support of multiple aspects weaving on the same point and *(iii)* the use of multiple activities as potential target of an aspect. However, the approach is dedicated to sequence diagrams (and the associated structural concerns described in class diagrams) which cannot be used "as is" to represent business processes. Finally, the composition operator defined in the approach relies on activity ordering, and does not support naturally the activity parallelism. Thus, Adore defines a domain–specific (Soa) composition mechanisms, dedicated to the intrinsic roots of business processes.

8. Conclusions

In this paper, we propose a meta–model named Adore that reifies key concepts associated to business process and then adequately support their evolution. We propose a logic–based meta–model, where process activities are represented as nodes, and relations between activities (*e.g.*, wait or guards) as edges. The meta–model handles concepts dedicated to support a compositional approach of business process design, with the definition of *fragment* of processes. These fragments are incomplete by nature, and aim to be integrated into others processes. The Weave algorithm supports the evolution of processes through the integration of fragments into other processes. An interesting characteristic of this algorithm is its order–independence: the obtained result does not depend on the execution order of the weave directives expressed by the designers. As a consequence, it is intrinsically different (but hopefully complementary) to sequential approaches such as features or aspects.

Finally, we expose as perspectives several research topics identified as perspectives of Adore. Some of them are based on ongoing collaborations with other universities, but most of them are open fields which may lead to very exciting research dealing with the evolution of business processes through a compositional approach.

Introducing semantics. Adore does not reify the semantic associated to each behavior. As a consequence, the interference detection mechanisms will only detect syntactic interferences. Coupled with a semantic web engine to express the relationships between services and exchanged data, an important part of the interference resolution may be automated. This idea is inspired by the one described in [41], where the authors automatically repair data–flow in Bpel processes, based on semantic annotations.

Action sequence optimization. An a–priori analysis of the generated action sequences may effectively reduce the length of these artifacts. Another direction is the optimization of sequences based on action semantics: if an element e is going to be deleted with the execution of the action a_i, one may consider to not execute actions which modifies e before its deletion (at the end, it will be deleted, so why should we waste time on this element?). These optimizations are not that easy [9], but may lead to an approximation of minimal sequences.

Execution optimization. Nowadays, multi–core architecture are usual, even on personal computer. The use of multiple threads in an application can *speedup* its execution time. In the context of ADORE, the action sequence to be executed can be analyzed to identify disjoint actions (that is, actions which do not work on the same artifacts). Consequently, a thread pool can be used to introduce parallelism inside the execution engine. Following the same idea, the computation of action sequences associated to composition directives are intrinsically independent, and may be executed in parallel (even automatically [42]). Preliminary results show a real benefit (one–to–one, *i.e.*, approximatively two times faster on a dual–core processor) to introduce multi–threading in the composition engine, at the implementation level.

Dynamic weaving. The ADORE approach is static, supporting the definition of behavioral models, and their evolution at the model level. But the emerging "Models at Runtime" [43] paradigm emphasizes the need to handle models compositions at runtime, following a dynamic approach. Moreover, context–aware application needs to be reconfigured at runtime, which induces a dynamic modification of their behavior when encountering a new context of execution. For all these reasons, we found very important to now think about ADORE as a dynamic composition engine, identifying what should be enhanced to support such an approach.

References

[1] OASIS, Reference Model for Service Oriented Architecture 1.0, Tech. Rep. wd-soa-rm-cd1, OASIS (Feb. 2006).

[2] S. Mosser, G. Mussbacher, M. Blay-Fornarino, D. Amyot, From Aspect-oriented Requirements Models to Aspect-oriented Business Process Design Models, in: 10th international conference on Aspect Oriented Software Development (AOSD'11),, , ACM, Porto de Galinhas, 2011.

[3] OASIS, Web Services Business Process Execution Language Version 2.0, Tech. rep., OASIS (2007).

[4] S. Mosser, Behavioral Compositions in Service-Oriented Architecture, Ph.D. thesis, University of Nice, Sophia–Antipolis, France (Oct. 2010).

[5] S. Gregory, A Declarative Approach to Concurrent Programming, in: PLILP '97: Proceedings of the9th International Symposium on Programming Languages:

Implementations, Logics, and Programs, Springer-Verlag, London, UK, 1997, pp. 79–93.

[6] S. Mosser, M. Blay-Fornarino, ADORE: Graphical Syntax & Execution Semantics, Tech. Rep. R-2011-05-FR, I3S/Equipe MODALIS - Pôle GLC, Sophia Antipolis (Apr. 2011).

[7] S. Mosser, M. Blay-Fornarino, M. Riveill, Web Services Orchestration Evolution: A Merge Process For Behavioral Evolution, in: 2nd Eur. Conf. on Software Architecture (ECSA'08), Springer LNCS, Paphos, Cyprus, 2008, pp. 35–49.

[8] OMG, Meta Object Facility (MOF) Core Specification Version 2.0 (2006).

[9] X. Blanc, I. Mounier, A. Mougenot, T. Mens, Detecting Model Inconsistency through Operation-Based Model Construction, in: W. Schäfer, M. B. Dwyer, V. Gruhn (Eds.), ICSE, ACM, 2008, pp. 511–520.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, The MIT Press, 2001.

[11] K. E. Iverson, A Programming Language, John Wiley & Sons, Inc., New York, NY, USA, 1962.

[12] M. E. Stickel, A Unification Algorithm for Associative-Commutative Functions, J. ACM 28 (3) (1981) 423–434.

[13] M. Aksit, A. Rensink, T. Staijen, A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points, in: AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development, ACM, New York, NY, USA, 2009, pp. 39–50.

[14] J. Kienzle, N. Guelfi, S. Mustafiz, Crisis Management Systems, A Case Study for Aspect-Oriented Modeling, Requirements Document for TAOSD Special Issue, McGuill University & University of Luxembourg (Sep. 2009).

[15] S. Mosser, M. Blay-Fornarino, R. France, Workflow Design using Fragment Composition (Crisis Management System Design through ADORE), Transactions on Aspect-Oriented Software Development (TAOSD) Special issue on Aspect Oriented Modeling (2010) 1–34.

[16] J. Cardoso, Evaluating the Process Control-Flow Complexity Measure, in: ICWS, IEEE Computer Society, 2005, pp. 803–804.

[17] S. Mosser, C. Parra, L. Duchien, M. Blay-Fornarino, Using Domain Features to Handle Feature Interactions, in: Sixth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12), VaMoS, ACM, Leipzig, Germany, 2012, p. 10.

[18] ERCIM, ERCIM Working Group on Software Evolution Terminology, Tech. rep., ERCIM (2010).

[19] R. Douence, D. Le Botlan, J. Noyé, M. Südholt, Concurrent Aspects, in: GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering, ACM, New York, NY, USA, 2006, pp. 79–88.

[20] A. Charfi, M. Mezini, Aspect-Oriented Web Service Composition with AO4BPEL, in: European Conference on Web Services, 2004, pp. 168–182.

[21] C. Courbis, A. Finkelstein, Towards Aspect Weaving Applications, in: ICSE '05: Proceedings of the 27th international conference on Software engineering, ACM, New York, NY, USA, 2005, pp. 69–77.

[22] J. Whittle, P. K. Jayaraman, A. M. Elkhodary, A. Moreira, J. Araújo, MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation, T. Aspect-Oriented Software Development VI 6 (2009) 191–237.

[23] R. Heckel, J. M. Küster, G. Taentzer, Confluence of Typed Attributed Graph Transformation Systems, in: ICGT '02: Proceedings of the First International Conference on Graph Transformation, Springer-Verlag, London, UK, 2002, pp. 161–176.

[24] S. Ciraci, W. Havinga, M. Aksit, C. Bockisch, P. van den Broek, A Graph-Based Aspect Interference Detection Approach for UML-Based Aspect-Oriented Models, T. Aspect-Oriented Software Development 7 (2010) 321–374.

[25] M. Clavreul, S. Mosser, M. Blay-Fornarino, R. France, Service–oriented Architecture Modeling: Bridging the Gap Between Structure and Behavior, in: MODELS'11, ACM/IEEE, Wellington, New Zealand, 2011, pp. 1–16.

[26] S. Ciraci, H. Sözer, M. Aksit, Guiding Architects in Selecting Architectural Evolution Alternatives, in: I. Crnkovic, V. Gruhn, M. Book (Eds.), ECSA, Vol. 6903 of Lecture Notes in Computer Science, Springer, 2011, pp. 252–260.

[27] M. Alférez, N. Amálio, S. Ciraci, F. Fleurey, J. Kienzle, J. Klein, M. E. Kramer, S. Mosser, G. Mussbacher, E. E. Roubtsova, G. Zhang, Aspect-Oriented Model Development at Different Levels of Abstraction, in: R. B. France, J. M. Küster, B. Bordbar, R. F. Paige (Eds.), ECMFA, Vol. 6698 of Lecture Notes in Computer Science, Springer, 2011, pp. 361–376.

[28] Workflow evolution, Data & Knowledge Engineering 24 (3) (1998) 211 – 238, eR'96. doi:10.1016/S0169-023X(97)00033-5.

[29] S. Rinderle, B. Weber, M. Reichert, W. Wild, Integrating process learning and process evolution - a semantics based approach, in: W. M. P. van der Aalst, B. Benatallah, F. Casati, F. Curbera (Eds.), Business Process Management, Vol. 3649, 2005, pp. 252–267.

[30] B. Weber, S. Rinderle, W. Wild, M. Reichert, Ccbr-driven business process evolution, in: Proceedings 6th International Conference on Case-Based Reasoning (ICCBR'06), Vol. 3620 of Lecture Notes in Computer Science, Springer Verlag, Berlin, 2005, pp. 610–624.

[31] W. M. P. van der Aalst, T. Basten, Inheritance of workflows: an approach to tackling problems related to change, Theor. Comput. Sci. 270 (2002) 125–203.

[32] S. Mosser, G. Hermosillo, A.-F. L. Meur, L. Seinturier, L. Duchien, Undoing event-driven adaptation of business processes, in: H.-A. Jacobsen, Y. Wang, P. Hung (Eds.), IEEE SCC, IEEE, 2011, pp. 234–241.

[33] Z. Hemel, L. C. L. Kats, E. Visser, Code generation by model transformation. A case study in transformation modularity, in: J. Gray, A. Pierantonio, A. Vallecillo (Eds.), International Conference on Model Transformation (ICMT08), Lecture Notes in Computer Science, Springer, 2008.

[34] F. Jouault, I. Kurtev, Transforming Models with ATL, in: J.-M. Bruel (Ed.), Satellite Events at the MoDELS 2005 Conference, Vol. 3844 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 128–138, 10.1007/11663430_14.

[35] F. Heidenreich, J. Kopcsek, U. Aßmann, Safe composition of transformations, Journal of Object Technology 10 (2011) 7: 1–20.

[36] U. Aßmann, Invasive Software Composition, Springer, 2003.

[37] H. Verjus, F. Pourraz, A Formal Framework For Building, Checking And Evolving Service Oriented Architectures, Fifth European Conference on Web Services ECOWS07 245–254doi:10.1109/ECOWS.2007.18.

[38] J. Klein, F. Fleurey, J.-M. Jézéquel, Weaving Multiple Aspects in Sequence Diagrams, T. Aspect-Oriented Software Development 3 (2007) 167–199.

[39] J. Kienzle, W. Al Abed, J. Klein, Aspect-Oriented Multi-View Modeling, in: AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development, ACM, New York, NY, USA, 2009, pp. 87–98.

[40] J. Klein, L. Hélouet, J.-M. Jézéquel, Semantic-based Weaving of Scenarios, in: proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06), ACM, Bonn, Germany, 2006.

[41] A. Moreau, J. Malenfant, M. Dao, Data Flow Repair in Web Service Orchestration at Runtime, in: M. Perry, H. Sasaki, M. Ehmann, G. O. Bellot, O. Dini (Eds.), ICIW, IEEE Computer Society, 2009, pp. 43–48.

[42] I. Dutra, D. Page, V. S. Costa, J. Shavlik, M. Waddell, Toward Automatic Management of Embarrassingly Parallel Applications, in: H. Kosch, L. Böszörményi, H. Hellwagner (Eds.), Euro-Par 2003 Parallel Processing, Vol. 2790 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2004, pp. 509–516, 10.1007/978-3-540-45209-6_73.

[43] D. Garlan, B. Schmerl, Using Architectural Models at Runtime: Research Challenges, in: European Workshop on Software Architectures (EWSA), Springer-Verlag, 2004, pp. 200–205.

Appendix A. Formalization Using First–order Logic

Definition 1. ***Variables*** *($\mathcal{V}$). A variable $v \in \mathcal{V}$ is defined as a name, a type and a boolean value $isSet?$ used to distinguish scalars from data–sets. Both name and type are defined as ground terms (*i.e., *terms without any variables).*

$$v = (name, type, isSet?) \in (Ground \times Ground \times \mathbb{B}) = \mathcal{V}$$

We consider here the PicWeb process defined in Fig. 1(a). The process receives a *tag* from the user and build a set of pictures (represented by their URLs) $pict^\star$ as an output.

$$tag = (tag, string, false) \in \mathcal{V},\ \ pict^\star = (pict^\star, url, true) \in \mathcal{V}$$

Definition 2. ***Activities*** ($\mathcal{A}$). *An activity $a \in \mathcal{A}$ is defined as a name, a kind $\in$ Kinds, a set of input variables inputs and a set of output variables outputs. The* Adore *meta–model defines nine Kinds of activities. We use closed terms (*i.e.*, terms without any free variables) to represent these concepts in first–order logic. The complete mapping between meta–model concepts and closed–terms is summarized in* Tab. *A.2*

$$a \quad = \quad (name, kind, ins, outs) \in (Ground \times Kinds \times \mathcal{V}^\star \times \mathcal{V}^\star) = \mathcal{A}$$

Intention	Bpel Concept	Term/Arity	Syntax
Variable assignment	`Assign`	$assign/1$	$assign(fct)$
Service invocation	`Invoke`	$invoke/2$	$invoke(srv, op)$
Synchronization	`Nop`	$nop/0$	nop
Message reception	`Receive`	$receive/0$	$receive$
Response sending	`Reply`	$reply/0$	$reply$
Fault throwing	`Throw`	$throw/0$	$throw$
Predecessors	*n/a*	$\mathbb{P}/0$	$\mathbb{P}$
Hook	*n/a*	$hook/0$	$hook$
Successors	*n/a*	$\mathbb{S}/0$	$\mathbb{S}$

Table A.2: *Kind*: using closed terms to specialize activities

The PicWeb process depicted in Fig. 1(a) contains the following activities:

$$\begin{aligned} a_0 &= (a_0, receive, \emptyset, \{tag\}) \\ a_1 &= (a_1, invoke(registry, getByKey), \{flickr\}, \{key\}) \\ a_2 &= (a_2, invoke(flickr, getPictures), \{key, tag\}, \{pict^\star\}) \\ a_3 &= (a_3, reply, \{pict^\star\}, \emptyset) \end{aligned}$$

Definition 3. ***Relations*** ($\mathcal{R}$). *Relations defined in* Adore *are used to model activity scheduling. We define a binary relation as an ordered pair of activities (*i.e.*, left and right). We denote as $left < right$ the fact that a relation exists between left and right. As* Adore *defines different kinds of relations, we use a label (defined as a closed term) to reify this information.*

$$left < right \quad \Rightarrow \quad \exists r = (left, right, label) \in (\mathcal{A} \times \mathcal{A} \times Closed) = \mathcal{R}$$

In this section, we describe the semantics of each relation *in isolation*, considering only two activities. We focus on the two most important relations available in Adore, *i.e.*, *waitFor* and *guards*. The complete execution semantics associated to Adore is described in [6].

- Wait For ($a \prec a'$): it defines a basic wait between two activities. Let $a \prec a' \in \mathcal{R}$. The activity a' will start only after the end of the activity a.

- Guards ($a \overset{c}{\prec} a'$, $a \overset{\neg c}{\prec} a'$): It enforces the *waitFor* relation to restrict the execution of its *right* activity according to the boolean response of the *left* one. Since $a \overset{c}{\prec} a'$, the a' activity will start only if the end of activity a assigns the boolean *true* to the variable c. For the sake of simplicity, we use the $a \overset{c}{\prec} a'$ notation to refer to the existence of a guard, and the $a \overset{\neg c}{\prec} a'$ notation to refer to its opposite.

In the PicWeb process described in Fig. 1(b), the wait between a_0 and c_1 is modeled as $a_0 \prec c_1 = (a_0, c_1, waitFor)$. The conditional wait between c_1 and c_3 (we only read the cache when it is valid, *i.e.*, when c is evaluated to *true*) is modeled as $c_1 \overset{c}{\prec} c_3 = (c_1, c_3, guard(c))$. As the $<$ symbol represents *any* relation, both $a_0 < c_1$ and $c_1 < c_3$ are true.

Definition 4. ***Path (*****i.e.,** ***relation transitive closure)****. We define a path between two activities a and a′ as the set of related activities used to go from a to a′ according to a given set of relations $R \in \mathcal{R}^\star$. The function path$^+$ (defined as path extension) computes all the existing paths between a and a′.*

$$\begin{array}{rcl} path : \mathcal{R}^\star \times \mathcal{A} \times \mathcal{A} & \to & \mathcal{A}^\star \\ (R, a, a') & \mapsto & a^\star \equiv \begin{cases} a < a' \in R \Rightarrow a^\star = \{a, a'\} \\ \wedge\ \exists a < \alpha \in R,\ path(R, \alpha, a') = A \\ \quad \Rightarrow a^\star = \{a\} \cup A \end{cases} \\ path^+ : \mathcal{R}^\star \times \mathcal{A} \times \mathcal{A} & \to & \{\mathcal{A}^\star\}^\star \\ (R, a, a') & \mapsto & \{A^\star \mid \forall p = path(R, a, a'), p \in A^\star\} \end{array}$$

Definition 5. ***Business Processes ($\mathcal{P}$)****. A business process $p \in \mathcal{P}$ is defined as a name (defined as a ground term), a set of variables (vars) to exchange data between service invocations, a set of activities (acts) which manipulate the variables, and a set of relations (rels) to schedule the activity set according to a partial order. Considering activities as vertices and relations as arcs, we can map a process to a directed graph.*

$$p \quad = \quad (name, vars, acts, rels) \in (Symbol \times \mathcal{V}^\star \times \mathcal{A}^\star \times \mathcal{R}^\star) = \mathcal{P}$$

The process depicted in Fig. 1(a) is then modeled as the following:

$$v_0 = (v_0, \{tag, flickr, key, pict^\star\}, \{a_0, a_1, a_2, a_3\}, \{a_0 \prec a_1, a_1 \prec a_2, a_2 \prec a_3\})$$

For a given process $p \in \mathcal{P}$, we refer to the previously defined *path* function according to the $rels(p)$ set of relations, and activities contained in the same process. Given a and a' two activities contained in $acts(p)$, we denote as $a \to a'$ the existence of a path between a and a'. We assume that $a = a' \Rightarrow a \to a'$.

$$a \to a' \equiv \exists p \in \mathcal{P}, a \in acts(p), a' \in acts(p),\ path(rels(p), a, a') \neq \emptyset\ \vee\ a = a'$$

Orchestrations $\neq$ Fragments. A business process can be either an orchestration *or* a fragment of orchestration. We define two boolean functions ($\mathcal{P} \to \mathbb{B}$)

named *isOrchestration*? and *isFragment*? to model this information. Fragments (denoted as $\mathcal{F}$) and orchestrations (denoted as $\mathcal{O}$) represent a partition of $\mathcal{P}$. These two sets can be defined in intention:

$$\mathcal{O} \;\equiv\; \{p \in \mathcal{P} \mid isOrchestration?(p)\}, \qquad \mathcal{F} \;\equiv\; \{p \in \mathcal{P} \mid isFragment?(p)\}$$
$$\mathcal{P} = \mathcal{O} \cup \mathcal{F}, \qquad \mathcal{O} \cap \mathcal{F} = \emptyset$$

Fragments specificities. Fragments represent incomplete behavior (which aims to be integrated into other processes) and consequently defines *ghosts* variables (handled in the fragment but concretely defined in the targeted business process). These variables are reified through fragment *hook* activity, which makes the link between the fragment and the targeted process. We define a dedicated function $ghost : \mathcal{F} \rightarrow \mathcal{V}^\star$ to identify such variables. To reify activities existing in the targeted process, we use the $\mathbb{P}$ (predecessors), $\mathbb{S}$ (successors) and *hook* activity kind. Note that a predecessor (resp. successor) activity can refer to a range of activities in the targeted process and a *hook* activity can refer to a block of activities. We also provide a set of eponymous functions to easily access to these artifacts in a given process:

$$e.g., \quad \mathbb{P} : \mathcal{F} \rightarrow \mathcal{A}, \qquad \mathbb{P}(f) = a \Rightarrow a \in acts(f) \wedge kind(a) = \mathbb{P}$$

Appendix B. Consistency Rules as Logical Predicates

We present in this section a set of *properties* defined over the previously defined formal concepts. An Adore model (*i.e.*, a model conforms to the Adore meta–model) assumes the respect of these properties[15]. These properties reify constraints expressed over the meta–model and consequently restrict the Adore expressiveness into entities compatible with the evolution mechanism described in the upcoming sections.

Property 4. ***(variable uniqueness)****. Let $v \in \mathcal{V}$ a variable. There is no other variables which use the same name. This property allow the assimilation of a variable and its name while performing a composition, and avoid* inadvertent capture*:* $\forall (v, v') \in \mathcal{V}^2,\ name(v) = name(v') \Rightarrow v = v'$

Property 5. ***(activity uniqueness)****. Let $a \in \mathcal{A}$ an activity. There is no other activity which use the same name. This property allow the assimilation of an activity and its name while performing a composition.*

$$\forall (a, a') \in \mathcal{A}^2,\ name(a) = name(a') \Rightarrow a = a'$$

Property 6. ***(restriction on fragment activities)****. Let $u \in \mathcal{U}$, and $f \in frags(u)$ a fragment. As* predecessors *and* successors *potentially represent multiple activities dealing with multiple variables, they cannot hold any variable from the fragment point of view:* $\forall a \in \mathcal{A},\ kind(a) \in \{\mathbb{P}, \mathbb{S}\} \Rightarrow vars(a) = \emptyset$

[15]The implemented engine throws an exception when it identifies a constraint violation.

Property 7. ***(relation containment)***. *A relation contained in a process $p \in \mathcal{P}$ involves activities from the same process.*

$$\forall a < a' \in \mathcal{R}, \exists! p \in \mathcal{P}, a < a' \in rels(p) \Rightarrow a \in acts(p) \wedge a' \in acts(p)$$

Property 8. ***(acyclic relations)***. *Relations in* ADORE *reify* waits *between activities. As it does not make any sense for an activity to wait for its own end, all relations defined in* ADORE *are defined as irreflexive and asymmetric. As a consequence, a process $p \in \mathcal{P}$ is assimilated as a directed acyclic graph (DAG).*

$$\begin{array}{ll} \textit{Irreflexive relations:} & \forall a \in \mathcal{A},\ a < a \notin \mathcal{R} \\ \textit{Asymmetric relations:} & \forall (a, a') \in \mathcal{A}^2, a < a' \in \mathcal{R} \Rightarrow a' < a \notin \mathcal{R} \\ \textit{Acyclic processes:} & \forall (a, a') \in \mathcal{A}^2, a < a' \in \mathcal{R} \Rightarrow a' \to a \notin \mathcal{R} \end{array}$$

Property 9. ***(single entry–point)***. *Let $p \in \mathcal{P}$. There is only one unique entry point activity $a \in acts(p)$ defined in p: $\forall p \in \mathcal{P}, \exists! a \in acts(p), a = entry(p)$*

Property 10. ***(activity path completeness)***. *All activities defined in a process $p \in \mathcal{P}$ must be connected to the entry point and to at least one exit point.*

$$\begin{array}{l} \forall p \in \mathcal{P}, \forall a \in acts(p), a \neq entry(p), a \notin exit(p) \\ \quad \Rightarrow \exists e \in exit(p), first(p) \to a \wedge a \to e \end{array}$$

Property 11. ***(fragment structure)***. *Let $f \in \mathcal{F}$ a fragment. This fragment can only define one* predecessor *activity ($\mathbb{P}$), one* hook *activity and one* successors *activity ($\mathbb{S}$).*

$$\begin{array}{l} \forall f \in \mathcal{F},\ \exists! a_p \in acts(f), a_p = \mathbb{P}(f) \\ \forall f \in \mathcal{F},\ \exists! a_h \in acts(f), a_h = hook(f) \\ \forall f \in \mathcal{F},\ \exists! a_s \in acts(f), a_s = \mathbb{S}(f) \end{array}$$

Appendix C. Execution Semantics

The execution semantics is based on boolean logic, as we associate to each activity a boolean formula computed over the set of its predecessors. The life–cycle of an activity can be formally defined with a deterministic automata $A = (Q, \Sigma, q_o, F)$ which defines four states: an activity *wait*s before being executed (initial state), *execute*s its internal behavior, *end*s its execution (*i.e.*, successful execution, final state) or *fail*s (*i.e.*, errorneous execution, final state). The alphabet of the automaton is as reduced as possible ($trigger$, $successful$ and $error$), and the associated transition function is deterministic. As a consequence, the life-cycle automata only accepts a language $\Lambda(A)$ containing two words (*i.e.*, successful or erroneous execution).

$$\begin{array}{rcl} Q & = & \{wait, execute, end, fail\} \\ \Sigma & = & \{trigger, successful, error\} \\ \Lambda(A) & = & \{"trigger, successful", "trigger, error"\} \end{array} \qquad \begin{array}{rcl} q_0 & = & wait \\ F & = & \{end, fail\} \end{array}$$

Each invocation of a process $p \in \mathcal{P}$ is executed as a different instance. When the process is invoked, we attach an automata to each activity defined in $acts(p)$. Each automata starts in its initial state, and waits for a *trigger* event to appear. Such a trigger is specific for each activity a, as it depends on the partial order defined by $rels(p)$. We model it as the satisfiability of a logical formula $\varphi(a)$. This formula composes the final states of a' predecessors, according to $rels(p)$. As soon as the system can satisfy $\varphi(a)$, the *trigger* symbol is sent to the automaton associated to a. Intuitively, the predecessors are combined with as a conjunction of their end. Special cases are handled by the introduction of the associated disjunctions, *e.g.*, exclusive predecessors of f_1 in FIG. 1(b).

Entry & Exit points. According to their kinds and their positions in the partial order induced by the relations, activities can be defined as entry or exit points. These definitions are used to verify the consistency of business process during and after the evolution.

$$\begin{array}{rcl} entry : \mathcal{P} & \rightarrow & \mathcal{A} \\ p & \mapsto & a \in acts(p), kind(a) \in \{receive, \mathbb{P}\} \wedge \nexists a' \in acts(p), a' \rightarrow a \\ exit : \mathcal{P} & \rightarrow & \mathcal{A}^{\star} \\ p & \mapsto & \{a \mid a \in acts(p), kind(a) \in \{reply, throw, \mathbb{S}\} \\ & & \quad \wedge \nexists a' \in acts(p), a \rightarrow a'\} \end{array}$$