ADORE: GRAPHICAL SYNTAX & EXECUTION SEMANTICS

Sébastien Mosser, Mireille Blay-Fornarino

Equipe MODALIS

Rapport de recherche
ISRN I3S/RR–2011-05–FR

Avril 2011
ABSTRACT:

The Service-oriented architecture (SOA) paradigm advocates the design of complex systems through the assembly of elementary services. From a behavioral point of view, these assemblies are called "business processes", designed at a coarse-grained level by domain experts. These artifacts aim to capture the business-driven core of the system. As the "business" of a company is not etched in stone, business processes must evolve per se. These enhancements include both business (e.g., new strategic partnership) and technical concerns (e.g., infrastructure change). However, all these concerns are tangled in the final process, defined as a monolithic artifact. In this context, we define the ADORE approach the tame this complexity. This technical report describes the ADORE meta-model, and its associated graphical syntax. It also provides an in-depth vision of the associated execution semantics. Intensively based on the fifth chapter of the associated thesis, our objective here is to provide a self-contained summary of these notions, independent of the thesis manuscript.

KEY WORDS:
Abstract

The Service-oriented architecture (SOA) paradigm advocates the design of complex systems through the assembly of elementary services. From a behavioral point of view, these assemblies are called “business processes”, designed at a coarse-grained level by domain experts. These artifacts aim to capture the business-driven core of the system. As the “business” of a company is not etched in stone, business processes must evolve per se. These enhancements include both business (e.g., new strategic partnership) and technical concerns (e.g., infrastructure change). However, all these concerns are tangled in the final process, defined as a monolithic artifact. In this context, we define the ADORE approach to tame this complexity. This technical report describes the ADORE meta-model, and its associated graphical syntax. It also provides an in-depth vision of the associated execution semantics. Intensively based on the fifth chapter of the associated thesis, our objective here is to provide a self-contained summary of these notions, independent of the thesis manuscript.

Contents

1 Introduction 2
2 The ADORE Meta-model 2
3 Graphical Syntax 5
  3.1 Representation of Activities ........................................ 5
  3.2 Representation of Relations ........................................ 7
  3.3 Iteration Policy ...................................................... 7
  3.4 Process Representation ............................................. 7
4 Execution Semantics 8
  4.1 Activity Lifecycle Automaton ....................................... 9
  4.2 Process Execution & Activity Triggering .......................... 10
  4.3 Iteration Policies Handling ........................................ 10
5 From Relations to Trigger Formula (ϕ) 11
  5.1 Process Entry Point & Single Predecessors ......................... 11
  5.2 Composition of Relations .......................................... 11
6 Iteration Policy Handling 17
7 Conclusions 20
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 [9]. Services are loosely-coupled by definition, and complex services are built upon basic ones using composition mechanisms. These compositions describe how services will be orchestrated when implemented, and are typically created by business process specialists. The loose coupling methodology enables the separation of concerns and helps system evolution. Using Web Services as elementary services, and Orchestrations [11] as composition mechanisms, Web Service Oriented Architectures (WSOA) provide a way to implement these loosely-coupled architectures.

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 need to address multiple concerns in the same artifact. For a given process, a business expert designs its own process, which is then enriched by concerns added by the security expert, the persistence expert, the sales expert, ...

We define the ADORE meta-model [8] to tame this complexity, supporting designers while they define complex business processes. The objective of this technical report is to provide a comprehensive vision of ADORE (self-contained). Based on a model-driven description of the meta-model, we provide here a description of the graphical syntax and the associated execution semantics. This work is intensively based on the fifth chapter of the associated thesis.

2 The ADORE Meta-model

The ADORE meta-model defines an Universe as its root. This concept contains all the business processes available for the composition algorithms. We define a BusinessProcess as a set of variables, a set of activities and a set of relations reifying a partial order between the activities. A business process can define two different kinds of artifacts: (i) as Orchestration (which realizes the behavior associated to an operation of service) or (ii) a Fragment (defining an incomplete behavior which aims to be integrated into another process). The associated graphical representation is depicted in Fig. 1.

![Figure 1: Global overview of the ADORE meta-model](image-url)
The Activity concept represents an elementary task realized by a given business process. An activity uses a set of inputs variables, and assigns its result in a set of outputs variables. The different types of activities that can be defined in ADORE (see Fig. 2) include (i) service invocation (denoted by Invoke), (ii) variable assignment (Assign), (iii) fault reporting (Throw), (iv) message reception (Receive), (v) response sending (Reply), and (vi) the null activity, which is used for synchronization purpose (Nop). Consequently, the ADORE meta-model contains all the atomic activities defined in the BPEL normative document except the wait (stopwatch activity) and the rethrow

Figure 2: ADORE class hierarchy reifying activities kinds

ADORE defines four different types of relationships between activities (see Fig. 3). All relations are expressed using binary operators. ADORE can define simple wait between activities through the WaitFor operator. Using WeakWait, one can reify predecessors’ disjunction [3], to allow non-determinism in the order of events. Conditional branches in a process control-flow are represented using Guard relations. Finally, a fault catching mechanism is defined through the OnFailure relations. As the ADORE meta-model does not define composite activities, BPEL composite constructions are reified using the different relations available in the meta-model. A sequence of activities is defined by a WaitFor relation; If/then/else flows are modeled using Guard relations.

We represent data exchanged between activities using the Variable concept (see Fig. 4). Additionally to its name and type, a Variable also holds a boolean attribute isSet, determining if the entity is representing a set or a simple scalar. Constants are defined as extensions of simple variables, and contain an immutable value. Contrarily to the BPEL, we do not define control-loops (e.g., for, while) in ADORE. We represent iteration in ADORE using a “for each” mechanism (which can be executed concurrently or in sequence). An Activity may be bound to a Variable through an IterationPolicy. As a consequence, the activity will be executed “for each” data defined in the associated set.

The ADORE purpose is to support the evolution of orchestrations. To perform such a goal, we extend the previously defined meta-model to represent incomplete processes. Such processes (called Fragment, Fig. 5) 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.

1In 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.
Figure 3: ADORE class hierarchy reifying relations kinds

Figure 4: ADORE concepts associated to Variables and Constants
According to the ERCIM working group on software evolution, aspect-oriented approaches rely at a syntactic level on four elementary notions [1], identified as: (i) joinpoints, (ii) pointcuts (iii), advice and finally (iv) aspects. Joinpoints represent the set of well-defined places in the program where additional behavior can be added. In the ADORE context, 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. We reify advices as fragments. Finally, aspects are defined as a set of pointcuts and advices. There is no explicit aspect notion in ADORE. This is done according to the usage of composition algorithm [8].

3 Graphical Syntax

To make ADORE artifacts more readable, we define a graphical syntax associated to the formal model. The ultimate goal of this syntax is to allow one to easily understand the behavior of a given process. As ADORE is inspired by graph-theory, we use a graph-based representation. We basically represent activities as boxes, and relations between activities as arrows. We voluntarily choose to not represent variable declaration in the graphical syntax, according to our goal (understand the behavior).

3.1 Representation of Activities

We represent an ADORE activity as a box. The left part of the box contains the name of the activity. The right part of the box is dedicated to activity content, according to its kind and variable usage:

- Variables are represented by their name,
  - Data-sets \(v \in V, isSet(v)\) are postfixed with a *;
  - Constants \(c \in C\) are valued and surrounded by quotes.
- Sets of variables are represented using a parenthesized form \(v_1, \ldots, v_n\),

Figure 5: ADORE extension to integrate Fragment concept
- The := symbol denotes assignment,
- The srv::op notation denotes an invocation of the operation op exposed by the service srv,
- Special keywords are dedicated to their associated kind: receive, reply, nop & throw,
- Assignments are represented using their function as keyword.

The notation used for interface activities is depicted in Fig. 6, and the one associated to business activities is depicted in Fig. 7.

**Fragment–Specific Activities.** Fragments of processes define three special activities: (i) a hook, (ii) its predecessors and (iii) its successors. These activities refer to the behavior of the process where the fragment will be integrated to. We decide to represent these activities using a dashed notation instead of a plain one, to make the difference of their intrinsic nature explicit.

**Short Notation.** When the contents of an activity is not important for a given context, we allow the usage of simple boxes to represent an activity, without the contents right part. We use
\[ a = (\text{nop}, \emptyset, \emptyset) \equiv \begin{array}{c} a \quad \text{nop()} \equiv \begin{array}{c} a' \end{array} \end{array} \]

Figure 9: Syntactic shortcut used to lighten graphical representation

\begin{figure}
\begin{subfigure}{0.2\textwidth}
\centering
\begin{tikzpicture}
\node[shape=circle,draw,fill=black] (a) at (0,0) {a};
\node[shape=circle,draw,fill=black] (a1) at (0,-1) {a'};
\draw[->] (a) -- (a1);
\end{tikzpicture}
\caption{(a) \( a \prec a' \)}
\end{subfigure}
\begin{subfigure}{0.2\textwidth}
\centering
\begin{tikzpicture}
\node[shape=circle,draw,fill=black] (a) at (0,0) {a};
\node[shape=circle,draw,fill=black] (a1) at (0,-1) {a'};
\node[shape=circle,draw] (c) at (0,-1.5) {c};
\draw[->] (a) -- (c);
\draw[->, dashed] (c) -- (a1);
\end{tikzpicture}
\caption{(b) \( a \ll a' \)}
\end{subfigure}
\begin{subfigure}{0.2\textwidth}
\centering
\begin{tikzpicture}
\node[shape=circle,draw,fill=black] (a) at (0,0) {a};
\node[shape=circle,draw,fill=black] (a1) at (0,-1) {a'};
\node[shape=circle,draw,fill=black] (c) at (0,-1.5) {c};
\draw[->] (a) -- (c);
\draw[->, dashed] (c) -- (a1);
\end{tikzpicture}
\caption{(c) \( a \preceq a' \)}
\end{subfigure}
\begin{subfigure}{0.2\textwidth}
\centering
\begin{tikzpicture}
\node[shape=circle,draw,fill=black] (a) at (0,0) {a};
\node[shape=circle,draw,fill=black] (a1) at (0,-1) {a'};
\node[shape=diamond,draw,fill=red] (f) at (0,-1.5) {f};
\draw[->, red] (a) -- (f);
\end{tikzpicture}
\caption{(d) \( a \blacklozenge a' \)}
\end{subfigure}
\caption{Graphical representation associated to ADORE relations}
\end{figure}

This syntactic shortcut to illustrate ADORE capabilities when the activity kind is not relevant (e.g. activity scheduling). An example is represented in Fig. 9.

3.2 Representation of Relations

We represent \( a < a' \in R \) a relation between two activities as an arrow starting from \( a \) and ending in \( a' \). The ADORE meta–model defines four types of relations between activities. The label associated to a relation is represented by the graphic style applied to the arrow:

- a plain arrow represents \( \text{waitFor} \) relations,
- a plain arrow with an associated label represents a \( \text{guard} \),
- a dashed arrow with a round tail represents a \( \text{weakWait} \),
- a red arrow with a diamond tail represents a \( \text{onFailure} \) relation.

Example of these graphical notations can be found in Fig. 10

3.3 Iteration Policy

We illustrate in Fig. 11 the syntax associated to the iteration policy concept. We use a nested \( \text{sub–graph} \) notation, where activities represented inside the sub–graph are part of the activities involved in the iteration policy. The \( \text{kind} \) associated to the policy is represented enclosed by \( < > \). We use a \( \text{“x in x”} \) label defined inside the subgraph to represent an iteration policy \( I_p \) where \( \text{scalar}(I_p) = x \) and \( \text{set}(I_p) = x^* \).

3.4 Process Representation

We represent a \( \text{process} \) as a directed acyclic graph. Orchestrations and fragments differ according to their bottom label: an \( \text{orchestration} \) label is composed by the \( \text{extras} \) informations handled by the process, and a \( \text{fragment} \) label contains only the name of the fragment. To ease the identification of activities provenance in composed processes, we enhance fragment with \( \text{colors} \). As fragment–specific activities refer to an exterior behavior, they are not colored to emphasize the semantic difference. We use the same assumption to represent relation entering or leaving \( \text{successors} \) and \( \text{predecessors} \), represented as \( \text{dashed} \) lines (as they refer to relations defined in the \( \text{target} \) process).
4 Execution Semantics

In this section, we focus on the definition of a semantics associated to the execution of modelled activities. We will use this semantics in the next chapters to illustrate how the different composition algorithms interact with the original behavior during the composition. The goal of this section is (i) to describe the semantics associated to activity execution, and (ii) to illustrate how it can be instantiated for a given model.

The software architecture community defines several formal methods associated to the definition of architecture and their behavior. In the large family of Formal Architecture languages, one can use \( \pi \)-calculus \[13\] to model the architecture of software systems. The \( \pi \)-Diapason \[12\] approach is defined as a domain-specific \( \pi \)-calculus implementation, dedicated to orchestration of web services. Like the previous formalism, its goal is to check orchestration correctness by using a virtual machine in order to check its behavior.

Specification languages such as Al\(l\)oy \[5\] can be used to express complex behavior in a software system. After the definition of Signatures and Facts, one can express Predicates, Functions and Assertions used to perform model-checking on the defined model. The goal of Al\(l\)oy is to check model correctness, that is, the fact that the modeled behavior respects a given program specification. Al\(l\)oy can then exhibit a counter-example if such a behavior exists in the finite scope domain used to perform the check.

Petri-Nets can be applied to model workflows \[14\]. This formalism gives to workflow (i) a formal semantic, (ii) a graphical nature and (iii) a large expressiveness. Existing work on Petri-nets investigate properties associated to Petri-nets, as well as analysis techniques. However, according to Wil van der Aalst, “Petri nets describing real processes tend to be complex and extremely large. Moreover, the classical Petri net does not allow for the modeling of data and time”. As a consequence, the simple initial formalism must be enhanced into a more complex one in order to take care of these concerns.

Temporal Languages such as C\(c\)SL \[7\] allow one to express Clocks, and a quasi-order between clock events through the definition of precedence rules and exclusion between pairs of activities. Based on such a temporal description of a system, it is possible to simulate the execution and to analyze traces to check the correctness of a system against its specification. From a simplified point of view, C\(c\)SL can be seen as a domain-specific model-checker for time-driven models.

Finally, process algebra and the associated temporal logic implementation (such as the automata-driven approach Lt\(s\)a \[6\]) can be used to express the behavior of a system, as an automata. Temporal properties can be checked on the automata, such as deadlocks or famine. These languages focus on concurrency description, and provide a formal foundation (based on automata) for this point of view.
Synthesis. The previously described methods focus on model–checking and correctness validation. They reason on a system in–the–large, and do not focus on activities in isolation. However, our goal is to express the execution semantics associated to each activity of a process, as a basis to express and demonstrate properties kept during the composition of processes. As a consequence, we choose a simple formalism (that is, finite automaton and boolean logic) and illustrate the semantics using such a model.

Model–Driven Engineering & Executability. The Kermeta meta–language [2] is defined to “breathe life into meta–models”. Using an aspect–oriented approach, the language supports the enhancement of legacy meta–models. With this approach, one can implement new operations in a meta–model, and consequently add executability concerns in a structural meta–model. The Kermeta language is used as underlying support in the Ccsl tool. Using the same approach, it is possible to use Kermeta to make Adore models executable. However, this is not our goal here. We do not want to make Adore models executable, but instead predict their behavior in a static way. The main idea is to describe a–priori the semantic of an Adore model based on its description, where Kermeta will support runtime reasoning, on the fly.

4.1 Activity Lifecycle Automaton

The life–cycle of an activity includes several states. Informally, an activity \( a \in A \) waits for its execution to begin, then executes its internal logic before ending its execution in normal or erroneous state. We use a deterministic automata to model this life–cycle.

\[
\alpha = (Q, \Sigma, q_0, F)
\]

\(\Lambda(\alpha) = \{"trigger, successful"\} \cup \{"trigger, error"\}\)

Generalization for “Multiple Errors” Handling. According to the service–oriented paradigm, an operation may throw several kinds of faults. To support this feature, we extend the previously defined automata by modeling the fail state and the error symbol as closed terms of arity 1 (the fault name \( f \)). As a consequence, we can virtually represent an infinite number of errors, with a finite number of states. This is still valid in Adore semantic since the important information is that the activity ends in an error state.

\(^2\) A bridge between Adore and Ccsl is exposed as one of the perspectives of this thesis.
4.2 Process Execution & Activity Triggering

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 \( \text{acts}(p) \). 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 the relation set \( \text{rels}(p) \). We model it as the satisfiability of a logical formula \( \varphi(a) \). This formula composes the final states of \( a \)'s predecessors, according to \( \text{rels}(p) \). As soon as the system can satisfy \( \varphi(a) \), the trigger symbol is sent to the automaton associated to \( a \). The way \( \varphi(a) \) is computed according to \( \text{rels}(p) \) is described in Sec. 5.

Process End. A process instance is considered to be ended since one of its exit point reaches a final state. The final state of the process is then defined as the final state of the reached exit point. When a process instance is ended, it is automatically destroyed.

4.3 Iteration Policies Handling

When an activity \( a \in \mathcal{A} \) is involved in an iteration policy defined on a finite data-set \( d^* \equiv \{d_1, \ldots, d_n\} \), a must be executed for each \( d_i \). As a consequence, we associate for each \( d_i \in d^* \) an automaton \( \alpha_i \) to \( a \). We also enrich the trigger symbol of the automaton alphabet, now defined as a closed term of arity 1: \( \text{trigger}(i) \). Each automaton \( \alpha_i \) uses the trigger(i) symbol in its own alphabet \( \Sigma_i \). According to the same idea, the satisfaction of the formula \( \varphi_i(a') \) will push the trigger(i) symbol in the automaton.

\[
\begin{align*}
Q & = \{\text{wait, execute, end, fail(f)}\} \\
\Sigma & = \{\text{trigger, successful, error(f)}\} \\
\Lambda & = \{"\text{trigger, successful"}, "\text{trigger, error(f)}"\}
\end{align*}
\]

\[
\begin{array}{c|ccc}
Q \times \Sigma & \text{trigger} & \text{successful} & \text{error} \\
\hline
\text{wait} & \text{execute} & - & - \\
\text{execute} & - & \text{end} & - \\
\text{end} & - & - & - \\
\text{fail} & - & - & - \\
\end{array}
\]
5 From Relations to Trigger Formula ($\varphi$)

**Notation.** We denote as $\varphi(a)$ the trigger formula associated to $a$. This formula is defined as a boolean composition of the final states of $a$ predecessors. We define two syntactic shortcuts $\text{end}(a)$\(^3\) and $\text{fail}(a, f)$\(^4\) to reify these final states in $\varphi$. According to guards relations, boolean variable values may also be used. We denote as $v$ (respectively $\neg v$) the fact that the boolean variable $v \in V$ is valued with true (respectively false).

5.1 Process Entry Point & Single Predecessors

**Entry Point.** Let $p \in P$ a business process. The entry–point of $p$ starts automatically its execution, and does not need to wait for anything else. As a consequence, its trigger formula is defined as true, and immediately satisfied.

$$\forall a \in \text{acts}(p), \ a = \text{entry}(p) \Rightarrow \varphi(a) = \text{true}$$

**Single Predecessors.** Let $p \in P$ a business process. When an activity $a$ is preceded by a unique activity $a'$ according to $\text{rels}(p)$, the associated formula only depends on the type of relations between $a$ and $a'$.

- A waitFor implies to wait for the end of $a'$,
- A guard implies to wait for the end of $a'$ and check the condition value,
- An onFailure implies to wait for a given fault to be thrown.

$$\forall a \in \text{acts}(p), \exists! a' < a \in \text{rels}(p), \begin{cases} a' \prec a & \Rightarrow \varphi(a) = \text{end}(a') \\ a' \prec c \prec a & \Rightarrow \varphi(a) = \text{end}(a') \land c \\ a' \prec a & \Rightarrow \varphi(a) = \text{end}(a') \land \neg c \\ a' \blacktriangledown a & \Rightarrow \varphi(a) = \text{fail}(a', f) \end{cases}$$

These rules are illustrated in Fig. 13.

5.2 Composition of Relations

When several relations use the same activity as right part, we need to compose the left activities to obtain a consistent formula. The goal of this section is to formalize the intuitive semantics associated to the graphical representation presented above. The pressed reader may skip this formalization step and relies on his/her intuition. We consider here an activity $a \in A$, involved in a process $p \in P, a \in \text{acts}(p)$. To lighten the formula description, we use the $p$ notation to refer to such a business process. Before describing in details the different functions used to compute such a formula, we provide here an intuitive description of these functions. For a given activity $a$, the following situations must be handled while computing $\varphi(a)$:

- $\Phi_w$: weak predecessors reify an explicit disjunction in the entering flow,
- $\Phi_f$: an exceptional–path reaches the control–path from which it diverges, and then implicitly defines a disjunction. Nested fault are handled by a recursive function $\Phi_d$, dedicated to the identification of such disjunction,
- $\Phi_e$: predecessors guarded by exclusive conditions implicitly define a disjunction,
- $\Phi_r$: all others situations need to the conjunction of $a'$ predecessors. A dedicated function $\Phi_b$ reify the way formulas are computed for a single (binary) relation.

---

\(^3\)end($a$) $\equiv$ “the activity $a$ is in state end”

\(^4\)fail($a$, $f$) $\equiv$ “the activity $a$ is in state fail($f$)”
Composed Formula ($\Phi$). For a given activity $a$, we use this function to build the final formula $\varphi(a)$ associated to $a$. It consists of a logical formula computed on the preceding activities ($preds$ function) by the $\Phi_w$ function.

$$
\Phi : \mathcal{A} \rightarrow \text{Formula} \\
a \mapsto \varphi(a) = \Phi_w(a, \text{preds}(a)) \\
preds : \mathcal{A} \rightarrow \mathcal{A}^* \\
a \mapsto \{a' \mid \exists \, a' < a \in \text{rels}(p)\}
$$

Handling Weak Relations ($\Phi_w$). Using a $\text{weakWait}$ relation, one defines a disjunction in the predecessor set. According to this semantics, we compute the formula as the disjunction of such predecessors and a formula expressed on the rest of the predecessors ($\Phi_f$, see next paragraph). This function and the previous one are illustrated in FIG. 14.

$$
\Phi_w : \mathcal{A} \times \mathcal{A}^* \rightarrow \text{Formula} \\
(a, P) \mapsto (\lor_{a' \in \text{weaks}(a)} \text{end}(a')) \land \Phi_f(a, P \setminus \text{weaks}(a)) \\
\text{weaks} : \mathcal{A} \rightarrow \mathcal{A}^* \\
a \mapsto \{a' \mid \exists \, a' < a \in \text{rels}(p)\}
$$

Handling Fault Branches ($\Phi_f$). Predecessors of the activity can be defined as children of a “failure–fork”, that is, a point where the control–path and an exceptional path diverge. We call such a point an origin of failure$^5$ (the top–most point is computed through the $\text{origin}_f$ function).

$^5$FIG. 15 defines two origin of failure for the predecessor set $\{a, b, c, d, e, f, g\}$: $o_1$ and $o_3$ (since $o_2$ is surrounded by $o_1$, it is not considered at this step).
\[ \Phi(a) : \]
\[ \Phi(a) = \Phi_w(a, \text{preds}(a)) \]
\[ \text{preds}(a) = \{a_1, a_2, \ldots\} \]
\[ \Phi_w(a, \{a_1, a_2, \ldots\}) : \]
\[ P = \{a_1, a_2, \ldots\} \]
\[ \text{weaks}(a) = \{a_1, a_2\} \]
\[ \Phi_w(a, P) = (\text{end}(a_1) \lor \text{end}(a_2)) \land \Phi_f(a, \{\ldots\}) \]
\[ \Rightarrow \varphi(a) = (\text{end}(a_1) \lor \text{end}(a_2)) \land \Phi_f(a, \{\ldots\}) \]

**Figure 14:** Building \( \varphi(a) \): \( \Phi \) and \( \Phi_w \) illustration

Formulas associated to each branch (one per origin of failure, where such a “failure–driven” partition is obtained through the \( \Pi_f \) function) are conjuncted together, and then conjuncted with the formula associated to predecessors without an adequate origin of failure in their ancestors. This function is illustrated in Fig. 15.

\[ \Phi_f : \mathcal{A} \times \mathcal{A}^* \rightarrow \text{Formula} \]
\[ (a, P) \mapsto (\bigwedge_{b \in \Pi_f(a, P)} \Phi_d(a, b)) \land \Phi_d(a, P \setminus (\Pi_f(a, P))) \]
\[ \Pi_f : \mathcal{A} \times \mathcal{A}^* \rightarrow (\mathcal{A}^*)^* \]
\[ (a, P) \mapsto \{b \mid \exists o \in \text{origin}_f^+(P), b = \text{branch}(o, a)\} \]
\[ \text{origin}_f^+ : \mathcal{A}^* \rightarrow \mathcal{A}^* \]
\[ P \mapsto \{o \mid o = \text{origin}_f(P)\} \]
\[ \text{origin}_f : \mathcal{A}^* \rightarrow \mathcal{A} \]
\[ P \mapsto o, \exists o \uparrow x \in \text{rels}(p), \exists \alpha \in P, x \rightarrow \alpha \land \exists \alpha' \in \text{acts}(p), o' \rightarrow o, o' \neq o, \exists o' \in P, o' \downarrow \alpha' \]
\[ \text{branch} : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}^* \]
\[ (o, a) \mapsto \{a' \mid \exists a' < a \in \text{rels}(p), o \rightarrow a'\} \]

**Handling Disjunctive Flows (\( \Phi_d \)).** Considering a restricted subset of predecessors \( P \) (e.g., computed as a branch by \( \Phi_f \)), we need to identify the control–path (thanks to the function \( f_c \)) and the different exceptional–paths (see Fig. 16) existing in \( P \) ancestors (such a partition is obtained through the \( \Pi_e \) function). For each subset of activities, we inductively call \( \Phi_f \) to identify sub–branches and then associated formulas are built by induction. The stop–condition associated to this induction system is the reception of a subset of activities which only defines a control–path
\( \varphi(a') = \Phi(a') = \Phi_f(a', \{a, b, c, d, e, f, g\}) \quad (\text{weaks}(a') = \emptyset) \\
\Phi_f(a', \{a, b, c, d, e, f, g\}) : \\
\quad \Pi_f(a', \{a, b, c, d, e, f, g\}) = \{\{a, b, c, d, e\}, \{f, g\}\} \\
\quad \text{result} \Rightarrow \Phi_d(a', \{a, b, c, d, e\}) \land \Phi_f(a', \{f, g\}) \\
\Phi_d(a', \{a, b, c, d, e\}) : \\
\quad f_c(a', \{a, b, c, d, e\}) = \{a, b, c, e\} \quad (\neq \{a, b, c, d, e\}) \\
\quad \Pi_c(a', \{a, b, c, d, e\}) = \{\{d\}\} \\
\quad \text{result} \Rightarrow \Phi_f(a', \{a, b, c, d, e\}) \lor \Phi_e(a', \{d\}) \\
\Phi_f(a', \{a, b, c, d, e\}) : \\
\quad \Pi_f(a', \{a, b, c, e\}) = \{\{a, b\}\} \\
\quad \text{result} \Rightarrow \Phi_d(a, \{a, b\}) \land \Phi_d(a', \{c, e\}) \\
\Phi_d(a', \{a, b\}) : \\
\quad f_c(a', \{a, b\}) = \{a\} \quad (\neq \{a, b\}) \\
\quad \Pi_c(a', \{a, b\}) = \{\{b\}\} \\
\quad \text{result} \Rightarrow \Phi_f(a', \{a\}) \lor \Phi_e(a', \{b\}) \\
\quad \Rightarrow \text{end}(a) \lor \text{end}(b) \\
\Phi_d(a', \{c, e\}) : \\
\quad f_c(a', \{c, e\}) = \{c, e\} \quad (\neq \{c, e\}) \\
\quad \text{result} \Rightarrow \Phi_e(a', \{c, e\}) = \cdots = \text{end}(c) \land \text{end}(e) \\
\Phi_d(a', \{f, g\}) = \cdots = \text{end}(f) \lor \text{end}(g) \\
\Rightarrow \varphi(a) = (\text{end}(a) \lor \text{end}(b)) \land \text{end}(c) \land \text{end}(e) \lor \text{end}(d) \land (\text{end}(f) \lor \text{end}(g))$

Figure 15: building \( \varphi(a') \): \( \Phi_f \) and \( \Phi_d \) inductive system
(i.e., there is no existing sub-branch defined inside this subset).

\[ \Phi_d : \mathcal{A} \times \mathcal{A}^* \rightarrow \text{Formula} \]
\[
    (a, P) \mapsto \begin{cases} 
        f_c(a, P) = P \Rightarrow \Phi_e(a, P) \\
        f_c(a, P) \neq P \Rightarrow \Phi_f(a, f_c(a, P)) \vee (\bigvee_{p \in \Pi_e(a, P)} \Phi_e(a, p)) 
    \end{cases}
\]

\[ f_c : \mathcal{A} \times \mathcal{A}^* \rightarrow \mathcal{A}^* \]
\[
    (a, P) \mapsto \{ a' \mid a' \in P, \exists o \in \text{origin}_f(a, P), o \rightarrow a' \}
\]

\[ \Pi_e : \mathcal{A} \times \mathcal{A}^* \rightarrow (\mathcal{A}^*)^* \]
\[
    (a, P) \mapsto \{ \text{flow} \mid \exists o \in \text{origin}_f^+(P), \exists f \in \text{faults}(o), \text{flow} = f_c(a, P, o, f) \}
\]

\[ \text{faults} : \mathcal{A} \rightarrow \text{Term}^* \]
\[
    a \mapsto \{ f \mid \exists a \not\in \text{rels}(p) \}
\]

\[ f_c : \mathcal{A} \times \mathcal{A}^* \times \mathcal{A} \times \text{Term} \rightarrow \mathcal{A}^* \]
\[
    (a, P, o, f) \mapsto \{ a' \mid a' \in P, \exists o \not\in \text{rels}(p), o \rightarrow a' \}
\]

**Handling Exclusive Activities (\( \Phi_e \)).** We are considering now \( P \) a subset of \( a \) predecessors which does not contain any exclusivity induced by fault–catch mechanisms (inductively solved). Activities can be exclusive according to their transitive guards. Based on all the available guards expressed on \( P \) members (these conditions are obtained through the \( \text{conds} \) function), we compute the formula by using a dedicated function \( \Phi_X \).

\[ \Phi_e : \mathcal{A} \times \mathcal{A}^* \rightarrow \text{Formula} \]
\[
    (a, P) \mapsto \Phi_X(A, P, \text{conds}(P))
\]

\[ \text{conds} : \mathcal{A}^* \rightarrow (\mathcal{V})^* \]
\[
    P \mapsto \{ V \mid \forall a \in P, V = \text{guards}^+(a) \}
\]

\[ \text{guards}^+ : \mathcal{A} \rightarrow \mathcal{V}^* \]
\[
    a \mapsto \{ v \mid \text{guard}(a, v) \}
\]

\[ \text{guard} : \mathcal{A} \times \mathcal{V} \rightarrow \mathcal{E} \]
\[
    (a, v) \mapsto \exists g : \alpha \in \text{rels}(p), \alpha \rightarrow a, \exists g : \alpha' \in \text{rels}(p), \alpha' \rightarrow a \\
    \vee \exists g : \alpha \in \text{rels}(p), \alpha \rightarrow a, \exists g : \alpha' \in \text{rels}(p), \alpha' \rightarrow a
\]

If the given set of conditions \( C \) is empty, there is no possible exclusivity between activities, and the computation of the formula is delegated to the \( \Phi_e \) function which builds a conjunction of waits. In the other case, we start by picking one\(^6\) condition \( c \in C \), and remove\(^7\) it from \( C \). We compute a partition of the predecessors set, according to their relation with the \( c \) condition: (i) predecessors guarded by \( c = \text{true} \) (thanks to the \( \text{onTrue} \) function), (ii) predecessors guarded by \( c = \text{false} \) (thanks to the \( \text{onFalse} \) function) and finally (iii) predecessors with no relation with \( c \) (function \( \text{notOn} \)).

The formula built on this partition is by essence valid according to the guard vivacity property [8]. We built the final formula thanks to a recursive call on \( \Phi_X \), as the disjunction of formulas associated to \( \text{onTrue} \) and \( \text{onFalse} \) activities, conjuncted with the formula associated to the rest

---

\(^6\)We consider a function \( \text{one} : (X^*)^* \rightarrow X \) which picks an element in the set which has the minimum cardinality. For example, \( \text{one}\{\{a, b\}, \{a, b, c, d\}\} = a \)

\(^7\)We consider a function \( \text{remove} : X \times (X^*)^* \rightarrow (X^*)^* \) which removes an element in all the existing subsets. For example, \( \text{remove}(a, \{\{a, b\}, \{a, b, c, d\}\}) = \{\{b\}, \{b, c, d\}\} \)
$\Phi(a):$

preds(a) = \{a_3, f_{01}, f_{02}, f'_02, a_4\} = P

result $\leadsto \Phi_w(a, P) = \Phi_f(a, P) \quad (\text{weak}(a) = \emptyset)$

$\Phi_f(a, P):$

$\Pi_f(a, P) = \{\{a_3, f_{01}, f_{02}, f'_02\}\}$

result $\leadsto \Phi_d(a, \{a_3, f_{01}, f_{02}, f'_02\}) \wedge \Phi_d(a, \{a_4\})$

$\Phi_d(a, \{a_3, f_{01}, f_{02}, f'_02\})$:

$P' = \{a_3, f_{01}, f_{02}, f'_02\}$

$f_c(a, P') = \{a_3\} \quad (\neq P')$

$\Pi_c(a, P') = \{\{f_{01}\}, \{f_{02}, f'_02\}\}$

result $\leadsto \Phi_e(a, \{f_{01}\}) \lor \Phi_e(a, \{f_{02}, f'_02\}) \lor \Phi_f(a, \{a_3\})$

$\Rightarrow \varphi(a) = (\text{end}(f_{01}) \lor (\text{end}(f_{02}) \land \text{end}(f'_02)) \lor \text{end}(a_3)) \land \text{end}(a_4)$

Figure 16: Building $\varphi(a)$: Multiple Faults handling in $\Phi_d$
of the predecessors. An illustration of this function usage is depicted in Fig. 17.

\[ \Phi_x : \mathcal{A} \times \mathcal{A}^* \times (\mathcal{V}^*)^* \rightarrow \text{Formula} \]

\[ (a, P, C) \mapsto \begin{cases} C = \emptyset & \Rightarrow \Phi_c(a, P) \\ C \neq \emptyset & \Rightarrow F \end{cases} \]

Let \( c = \text{one}(C), C' = \text{remove}(c), \)

\[ F = \Phi_x(a, \text{notOn}(P, c), C') \land ( \Phi_x(a, \text{onTrue}(P, c), C') \lor \Phi_x(a, \text{onFalse}(P, c), C')) \]

\[ \text{onTrue} : \mathcal{A}^* \times \mathcal{V} \rightarrow \mathcal{A}^* \]

\[ (P, v) \mapsto \{ a \mid \exists g \prec \prec a \in \text{rels}(p), a \rightarrow a, \exists g \prec \prec a' \in \text{rels}(p), a' \rightarrow a \} \]

\[ \text{onFalse} : \mathcal{A}^* \times \mathcal{V} \rightarrow \mathcal{A}^* \]

\[ (P, v) \mapsto \{ a \mid \exists g \prec \prec a \in \text{rels}(p), a \rightarrow a, \exists g \prec \prec a' \in \text{rels}(p), a' \rightarrow a \} \]

\[ \text{notOn} : \mathcal{A}^* \times \mathcal{V} \rightarrow \mathcal{A}^* \]

\[ (P, v) \mapsto P \setminus (\text{onTrue}(P, v) \cup \text{onFalse}(P, v)) \]

**Handling Conjunctive Flows (\( \Phi_c \)).** Considering a subset \( P \) of a predecessors without any exclusive activities, the formula associated to \( P \) is a conjunction of all the waits induced by the binary relation existence (computed through \( \Phi_b \)).

\[ \Phi_c : \mathcal{A} \times \mathcal{A}^* \rightarrow \text{Formula} \]

\[ (a, P) \mapsto \bigwedge_{a' \in P} \Phi_b(a, a') \]

**Handling Binary Relation (\( \Phi_b \)).** According to the previous functions, we are now handling binary relations, and expect to transform it into a Formula. The result depends on the kind of relation defined in the relation label, as explained in Sec. 5.1.

\[ \Phi_b : \mathcal{A} \times \mathcal{A} \rightarrow \text{Formula} \]

\[ (a, a') \mapsto \begin{cases} a' \prec a \in \text{rels}(p) \Rightarrow \text{end}(a') \\ a' \prec a \in \text{rels}(p) \Rightarrow \text{end}(a') \land c \\ a' \prec a \in \text{rels}(p) \Rightarrow \text{end}(a') \land \neg c \\ a' \preceq a \in \text{rels}(p) \Rightarrow \text{fail}(a', f) \end{cases} \]

6 Iteration Policy Handling

We handle predecessors associated to an iteration policy \( I_p \) (defined on data-set \( d^* = \{d_1, \ldots, d_n\} \)) according to the kind of \( I_p \). We base the semantics of these policies on the BPEL definition of the **forEach** composite activity:

\[ ^* \text{ The } \text{<forEach>} \text{ activity will execute its contained } \text{<scope>} \text{ activity exactly } N+1 \text{ times where } N \text{ equals the } \text{<finalCounterValue>} \text{ minus the } \text{<startCounterValue>}. \]

- If the value of the parallel attribute is no then the activity is a serial **forEach**. The enclosed **scope** activity MUST be executed \( N+1 \) times, each instance starting only after the previous repetition is complete. [...] 

- If the value of the parallel attribute is yes then the activity is a parallel **forEach**. The enclosed **scope** activity MUST be concurrently executed \( N+1 \) times. [...] 

[10] The **forEach** activity without a **completionCondition** completes when all of its child **scope**’s have completed. [...] "
\[
\Phi(a) : \\
\text{preds}(a) = \{a_3, a_4, a_5, a_6, a_7\} = P \\
\text{result} \Rightarrow \Phi_\text{w}(a, P) = \Phi_\text{f}(a, P) = \Phi_\text{d}(a, P) = \Phi_\text{e}(a, P)
\]

\[
\Phi_\text{e}(a, P) : \\
\text{conds}(P) = \{\{c\}, \{c, c'\}\} = \text{Conds} \\
\text{result} \Rightarrow \Phi_\chi(a, P, \text{Conds})
\]

\[
\Phi_\chi(a, P, \text{Conds}) : \\
\text{one}(\text{Conds}) = c, \text{ remove}(c, \text{Conds}) = \{\{c'\}\} \\
\text{onTrue}(P, c) = \{a_5, a_6\}, \text{ onFalse}(P, c) = \{a_3, a_4\} \\
\text{notOn}(P, c) = \{a_7\} \\
\text{result} \Rightarrow (\Phi_\chi(a, \{a_5, a_6\}, \{\{c'\}\}) \lor \Phi_\chi(a, \{a_3, a_4\}, \{\{c'\}\})) \\
\quad \land \Phi_\chi(a, \{a_7\}, \{\{c'\}\})
\]

\[
\Phi_\chi(a, \{a_5, a_6\}, \{\{c'\}\}) : \\
\text{one}(\{\{c'\}\}) = c', \text{ remove}(c, \text{Conds}) = \emptyset \\
\text{onTrue}(\{a_5, a_6\}, c') = \emptyset = \text{onFalse}(\{a_5, a_6\}, c') \\
\text{notOn}(\{a_5, a_6\}, c') = \{a_5, a_6\} \\
\text{result} \Rightarrow \Phi_\chi(a, \{a_5, a_6\}, \emptyset) = \Phi_\text{e}(a, \{a_5, a_6\})_{\text{end}(a_5) \land \text{end}(a_6)}
\]

\[
\Phi_\chi(a, \{a_3, a_4\}, \{\{c'\}\}) : \\
\text{onTrue}(\{a_3, a_4\}, c') = \{a_3\}, \text{ onFalse}(\{a_3, a_4\}, c') = \{a_4\} \\
\text{notOn}(\{a_3, a_4\}, c') = \emptyset \\
\text{result} \Rightarrow \Phi_\chi(a, \{a_3\}, \emptyset) \lor \Phi_\chi(a, \{a_4\}, \emptyset) \\
\Rightarrow \Phi_\text{e}(a, \{a_3\}) \lor \Phi_\text{e}(a, \{a_4\})_{\text{end}(a_3) \land \text{end}(a_4)}
\]

\[
\Phi_\chi(a, \{a_7\}, \{\{c'\}\}) : \\
\text{onTrue}(\{a_7\}, c') = \emptyset = \text{onFalse}(\{a_7\}, c') \\
\text{notOn}(\{a_7\}, c') = \{a_7\} \\
\text{result} \Rightarrow \Phi_\chi(a, \{a_7\}, \emptyset) = \Phi_\text{e}(a, \{a_7\})_{\text{end}(a_7)}
\]

\[
\Rightarrow \varphi(a) = ((\text{end}(a_3) \lor \text{end}(a_4)) \lor (\text{end}(a_5) \land \text{end}(a_6))) \land \text{end}(a_7)
\]

Figure 17: Building \(\varphi(a)\): \(\Phi_\text{e}\) Illustration
Let $I_p$ an iteration policy defined on $d^*$. According to this definition, we express the following mapping between ADORÉ and BPEL concepts:

- **forEach** $\equiv I_p$
- **startCounterValue** $\equiv 1$
- **finalCounterValue** $\equiv |d^*|$
- **scope** $\equiv \text{acts}(I_p)$
- **parallel** $= \begin{cases} 
\text{yes} & \equiv \text{kind}(I_p) = \text{parallel} \\
\text{no} & \equiv \text{kind}(I_p) = \text{serial}
\end{cases}$
- **completionCondition** $= \emptyset$

We denote as end$_i(a)$ the fact that the automaton associated to the data $d_i$ for the activity $a$ reaches its end state. The $\Phi^i$ algorithm is defined as an enhancement of the previously described algorithm which propagates the $i$ concern in the computed formula for predecessors involved in an iteration policy. The behavioral distinction between serial and parallel policies can be defined in ADORÉ as the following:

- **Serial Policy:** The block contents is executed one by one, for each data. An illustration of this semantics is depicted in Fig. 19. It starts on the first data $d_1$, using the previously defined semantics to trigger the first activity ($S_2$). Until the last data $d_n$, reaching the end of the block for a data $d_i$ means triggering the $d_{i+1}$ execution ($S_2$). The last data $d_n$ triggers the execution of the block successors ($S_3$). The other activities use the previously defined semantics ($S_1$).

- **Parallel Policy:** The block contents is concurrently executed for each $d_i \in d^*$. An illustration of this semantics is depicted in Fig. 20. It starts by triggering all the first activities ($P_2$). Blocks successors wait for the satisfaction of the conjunction of all $\varphi_i$ formulas for the last activity of the block ($P_3$). The other activities use the previously defined semantics ($P_1$).

**Composition with the previously defined semantics.** Let $a \in \mathcal{A}$ an activity. To compute the complete execution semantic associated to $a$, we need to pre-process the predecessor set before using the previously defined algorithms. For each iteration policy associated to a predecessor of $a$, we compute the associated formula with a function $\Phi_{I^8}$. The previously defined algorithm $\Phi_w$ is then executed on the remaining predecessors. The final formula is built as a conjunction of these formulas. An illustration of this composition is depicted in Fig. 18.

**Limitation.** We do not handle error policies associated to iteration policies, and we consider that no fault can be thrown by activities defined in the activity block associated to a policy. This is coherent with the semantics associated to error-handling defined by the BPEL:

“If premature termination occurs such as due to a fault, [...] then this $N+1$ requirement does not apply.”

[10]

Considering iteration as equivalent to a final recursion, one can bypass this limitation, but must write the process using this style of writing (and rely on the execution engine to optimize the execution). Enhancing ADORÉ expressiveness to address in a better way error handling and error composition for iteration policies is exposed as one of the perspective of this work.

---

8Defined as an implementation of the previously described mechanisms
7 Conclusions

In this report, we described the meta-model used to support the ADORE approach. The ADORE meta-model allows designers to represent business processes (as orchestration of services) and fragments of business processes. We made the choice to only present in this document the formalism, and leave the implementation details (the meta-model and the associated tools are implemented in Prolog & Java and represent 13,000 lines of code) in the ADORE website (adore-design.org).

Thanks to this formalism, we described in a graphical syntax used to represent models of business processes. This syntax, inspired by UML activity diagrams, allows one to intuitively understand a process without priori knowledge on ADORE. However, we also described a formal execution semantic associated to the meta-model, anchoring this intuition into a computable set of logical formulas.

References


Definitions:

\[
\begin{align*}
\delta^* & = \{d_1, \ldots, d_n\}, \ |\delta^*| = n \\
B & = \{b_1, b_2, b_3, b_4, b_5\} \\
\text{firsts}(B) & = \{b_1, b_2\} = F \\
\text{nexts}(B) & = \{a', a''\} = N
\end{align*}
\]

General Description:

\[
\begin{align*}
(S_1) \quad & \forall a \in B \setminus F, \quad \varphi_1(a) = \Phi^i(a) \\
(S_2) \quad & \forall a \in F, \quad \varphi_1(a) = \Phi(a) \\
& \quad \varphi_{2 \leq i \leq n}(a) = \bigwedge_{\nu \in \text{nexts}} \Phi^{i-1}(\nu) \\
(S_3) \quad & \forall a \in N, \quad \varphi(a) = \Phi^N(a)
\end{align*}
\]

Computed Execution Semantics:

\[
\begin{align*}
\varphi(a) & = \text{true} \\
\varphi_1(b_3) & = \text{end}_i(b_1) \wedge \text{end}_i(b_2) \quad (S_1) \\
\varphi_1(b_4) & = \text{end}_i(b_3) \quad (S_1) \\
\varphi_1(b_5) & = \text{end}_i(b_3) \quad (S_1) \\
\varphi_1(b_1) & = \text{end}(a) \quad (S_2) \\
\varphi_{2 \leq i \leq n}(b_1) & = \text{end}_{i-1}(b_4) \wedge \text{end}_{i-1}(b_5) \quad (S_2) \\
\varphi_x(b_2) & = \varphi_x(b_1) \quad (S_2) \\
\varphi(a') & = \text{end}_N(b_4) \quad (S_3) \\
\varphi(a'') & = \text{end}_N(b_5) \quad (S_3)
\end{align*}
\]

Figure 19: Execution semantics associated to serial Iteration Policies
Definitions:

\[ d^* = \{d_1, \ldots, d_n\}, \quad |d^*| = n \]
\[ B = \{b_1, b_2, b_3, b_4, b_5\} \]
\[ \text{firsts}(B) = \{b_1, b_2\} = F \]
\[ \text{nexts}(B) = \{a', a''\} = N \]

General Description:

\[ (P_1) \forall a \in B \setminus F, \quad \varphi_i(a) = \Phi^i(a) \]
\[ (P_2) \forall a \in F, \quad \varphi_i(a) = \Phi(a) \]
\[ (P_3) \forall a \in N, \quad \varphi(a) = \bigwedge_{i=1}^{n} \Phi^i(a) \]

Computed Execution Semantics:

\[ \varphi(a) = \text{true} \]
\[ \varphi_i(b_3) = \text{end}_i(b_1) \land \text{end}_i(b_2) \quad (P_1) \]
\[ \varphi_i(b_4) = \text{end}_i(b_3) \quad (P_1) \]
\[ \varphi_i(b_5) = \text{end}_i(b_3) \quad (P_1) \]
\[ \varphi_i(b_1) = \text{end}(a) \quad (P_2) \]
\[ \varphi_i(b_2) = \text{end}(a) \quad (P_2) \]
\[ \varphi(a') = \bigwedge_{i=1}^{n} \text{end}_n(b_4) \quad (P_3) \]
\[ \varphi(a'') = \bigwedge_{i=1}^{n} \text{end}_n(b_5) \quad (P_3) \]

Figure 20: Execution semantics associated to parallel Iteration Policies