Vrije Universiteit Brussel

Semantics of the Unify Composition Mechanism

Technical Report SOFT-TR-2011.04.15

Niels Joncheere and Ragnhild Van Der Straeten
Vrije Universiteit Brussel
Software Languages Lab
Pleinlaan 2, 1050 Brussels, Belgium
{njonchee,rvdstrae}@vub.ac.be

Semantics of the UNIFY Composition Mechanism

Niels Joncheere and Ragnhild Van Der Straeten
Vrije Universiteit Brussel
Software Languages Lab
Pleinlaan 2, 1050 Brussels, Belgium
{njonchee,rvdstrae}@vub.ac.be

April 15, 2011

Abstract

Existing workflow languages have insufficient support for separation of concerns. This makes workflows hard to comprehend, maintain and reuse. The UNIFY framework addresses this problem by allowing to specify each workflow concern — regular or crosscutting — in isolation of other concerns, and providing a connector mechanism that is used to connect different concerns according to workflow-specific connection patterns. This technical report provides a detailed description of the semantics of the UNIFY connector mechanism by enumerating the graph transformation rules for each of its connector types.

Contents

1 History

Date	**Comment**
April 15, 2011	Creation

2 Introduction

Workflow management systems have become a popular technique for automating processes in many domains, ranging from high-level business process management to low-level web service orchestration. In each of these domains, workflows consist of several *concerns*. If all of a workflow's concerns can only be specified in a single, monolithic module, it will be hard to comprehend, maintain, or reuse the workflow. Therefore, mechanisms have been developed to decompose workflows into separate modules such as sub-workflows. Unfortunately, a workflow can only be decomposed according to one dimension, and concerns that do not align with this decomposition end up scattered across the workflow and tangled with one another.

This problem has been identified in software engineering research, and is called the *tyranny of the dominant decomposition* [14]. The concern that guides the decomposition is called the *base concern*, and concerns that do not align with it are called *crosscutting concerns*. Aspect-oriented programming (AOP) [9] is a well-known approach that allows modularizing crosscutting concerns in separate *aspects*. An aspect consists of a *pointcut* and an *advice*. A pointcut selects a set of locations — which are called *joinpoints* — in a program, and an advice specifies the concern's behavior, which should be executed *before*, *after*, or *around* each of these locations.

Existing research has identified the need for better separation of concerns in workflows: Charfi and Mezini [3] and Courbis and Finkelstein [4] have proposed aspect-oriented extensions to BPEL [1] that allow inserting functionality before, after, or around any BPEL activity. In our previous work on PADUS [2], we argued that workflows may require more than these classic before, after, and around advices. For example, it can be useful to allow adding an extra branch to a split, and this cannot be easily expressed using the classic advices. We implemented this new advice, and called it the *in* advice.

Our approach, which is called UNIFY, improves on existing research (including PADUS) on the following five points:

1. Existing research on modularization of workflow concerns is aimed at only modularizing crosscutting concerns [3, 4, 2], or at only modularizing one particular kind of concern, such as monitoring [5]. UNIFY, on the other hand, aims to provide a uniform approach for modularizing all workflow concerns.

2. Existing aspect-oriented approaches for workflows are fairly straightforward applications of general aspect-oriented principles, and are insufficiently focused on the concrete context of workflows. UNIFY improves on this by allowing workflow concerns to connect to each other in workflow-specific ways, i.e., the connector mechanism supports a number of dedicated control flow patterns.

3. UNIFY is designed to be applicable to a wide range of concrete workflow languages. This is accomplished by defining its connector mechanism in terms of a general base language metamodel.

4. UNIFY defines a clear semantics for both the workflow concerns and their connections. This facilitates the application of existing workflow verification techniques.

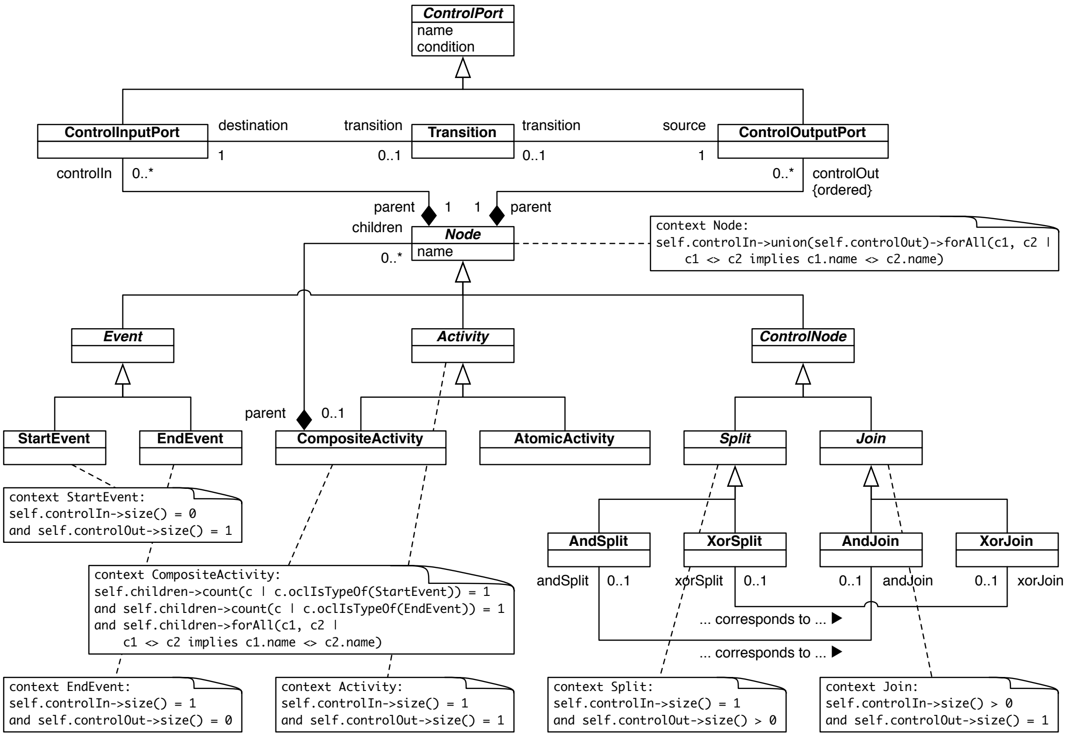
5. The UNIFY implementation can either be used as a separate workflow engine, or as a preprocessor that is compatible with existing workflow engines.

We already introduced the general concepts of our approach in previous work [6], without describing its concrete syntax, abstract syntax, semantics, or implementation. In [7], we provide a complete description of the UNIFY framework. However, space restrictions prevented us from giving a list of all graph transformation rules. This technical report does provide this information. Its structure is as follows. Section 3 gives a brief overview of UNIFY. Section 4 introduces graph transformation, and Section 5 enumerates our graph transformation rules. Section 6 concludes this technical report.

3 The UNIFY Framework

3.1 Base Language

At the heart of the UNIFY framework lies the base language meta-model shown in Figure 1.



```
context Node:
self.controlIn->union(self.controlOut)->forAll(c1, c2 |
    c1 <> c2 implies c1.name <> c2.name)
```

```
context StartEvent:
self.controlIn->size() = 0
and self.controlOut->size() = 1
```

```
context CompositeActivity:
self.children->count(c | c.oclIsTypeOf(StartEvent)) = 1
and self.children->count(c | c.oclIsTypeOf(EndEvent)) = 1
and self.children->forAll(c1, c2 |
    c1 <> c2 implies c1.name <> c2.name)
```

```
context EndEvent:
self.controlIn->size() = 1
and self.controlOut->size() = 0
```

```
context Activity:
self.controlIn->size() = 1
and self.controlOut->size() = 1
```

```
context Split:
self.controlIn->size() = 1
and self.controlOut->size() > 0
```

```
context Join:
self.controlIn->size() > 0
and self.controlOut->size() = 1
```

Figure 1: The UNIFY base language meta-model

A concern is modeled as a *CompositeActivity*. Each *CompositeActivity* has the following children:

- A *StartEvent*, which represents the point where the *CompositeActivity*'s execution starts.
- An *EndEvent*, which represents the point where the *CompositeActivity*'s execution ends.
- Any number of *Activities*, which are the units of work that are performed by the *CompositeActivity*.
- Any number of *ControlNodes*, which are used to route the *CompositeActivity*'s control flow.
- One or more *Transitions*, which connect the *StartEvent*, the *EndEvent*, the *Activities* and the *ControlNodes* to each other.

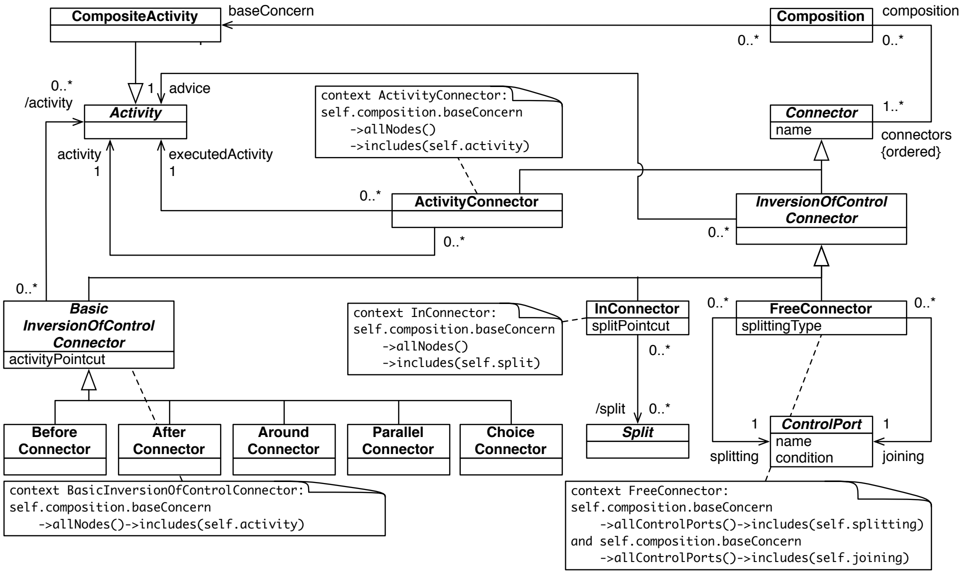
An *Activity* is either a *CompositeActivity* or an *AtomicActivity*. Nested *CompositeActivities* can be used to hierarchically decompose a concern, similar to the classic sub-workflow decomposition pattern. Each *Activity* has a name that is unique among its siblings in the composition hierarchy, and has one *ControlInputPort* and one *ControlOutputPort*. A *ControlInputPort* represents the point where control enters an *Activity*, while a *ControlOutputPort* represents the point where control exits an *Activity*. Each *ControlPort* has a name that is unique among its siblings. Within a *CompositeActivity*, the *StartEvent* is used to specify where the *CompositeActivity*'s execution should start when its *ControlInputPort* is triggered. The *EndEvent* is used to specify where the *CompositeActivity*'s execution should finish, and will cause the *CompositeActivity*'s *ControlOutputPort* to be triggered. Thus, a *StartEvent* only has a *ControlOutputPort*, and an *EndEvent* only has a *ControlInputPort*.

Transitions define how control flows through a *CompositeActivity*. This is done by connecting the *ControlOutputPorts* of the *CompositeActivity*'s *Nodes* to *ControlInputPorts*. *ControlNodes* can be used to route the flow of control, and are either *AndSplits*, *XorSplits*, *AndJoins* or *XorJoins*. A *Split* may have a corresponding *Join*. Together, *Transitions* and *ControlNodes* define a *CompositeActivity*'s control flow perspective.

The UNIFY base language meta-model does not aim to support every possible control flow pattern that has been identified in existing literature, as our research focuses on the expressiveness of the modularization mechanism rather than on the expressiveness of the individual modules. The UNIFY base language meta-model supports the *basic control flow patterns* [12], which are sufficient for expressing most workflows. We do not aim to support more advanced patterns such as cancellation and multiple instances. Due to the generic nature of the UNIFY base language meta-model, the cores of most workflow languages are compatible with it. We have extended the meta-model towards the cores of the WS-BPEL [8] and BPMN [10] workflow languages.

3.2 Connector Mechanism

UNIFY promotes separation of concerns by allowing workflow concerns to be specified in isolation of each other, as separate *CompositeActivities*. These can be executed by themselves, or can be connected to other concerns using connectors. Figure 2 shows the meta-model for the UNIFY connector mechanism. In the interest of brevity, the definition for the *CompositeActivity*'s *allNodes* and *allControlPorts* queries are omitted. These queries return the set of nodes and control ports, respectively, obtained by the transitive closure of the *children* relation.



```
context ActivityConnector:
self.composition.baseConcern
    ->allNodes()
    ->includes(self.activity)
```

```
context InConnector:
self.composition.baseConcern
    ->allNodes()
    ->includes(self.split)
```

```
context BasicInversionOfControlConnector:
self.composition.baseConcern
    ->allNodes()->includes(self.activity)
```

```
context FreeConnector:
self.composition.baseConcern
    ->allControlPorts()->includes(self.splitting)
and self.composition.baseConcern
    ->allControlPorts()->includes(self.joining)
```

Figure 2: The UNIFY connector meta-model

A *Composition* specifies which *CompositeActivity* is its base concern, and which *Connectors* are to be applied to it. The set of *Connectors* is ordered, and they will be applied according to this ordering.

Connectors can be used to add functionality at certain points in a concern. They can be divided into two categories: *ActivityConnectors* and *InversionOfControlConnectors*. In a traditional workflow language, a workflow can be divided into several levels of granularity through the use of sub-workflows. Control passes from the main workflow into sub-workflows and back, with the main workflow specifying when the sub-workflow should be executed. An *ActivityConnector* allows expressing that a certain *Activity* inside a certain concern should be implemented by executing another *Activity*, which thus acts as a sub-concern. By specifying this link in a separate connector instead of inside the concern, we reduce coupling between the concern and the sub-concern, thus promoting reuse.

InversionOfControlConnectors invert the traditional passing of control from main workflow into sub-workflows: they specify that a certain concern should be adapted, while this concern is not aware of this adaptation. In this way, such connectors can be used to add concerns that were not anticipated when the concern to which they are applied was created.

Joinpoints are well-defined points during the execution of a concern where extra functionality — the *advice* — can be inserted using an *InversionOfControlConnector*. Joinpoints in existing aspect-oriented approaches for workflows are either every XML element of the workflow definition [3, 4] or every workflow activity [2]. As is shown in Table 1, our approach supports three kinds of joinpoints: *Activities*, *Splits*, and *ControlPorts*.

Advice type	**Joinpoint**
before	*Activity*
after	*Activity*
around	*Activity*
parallel	*Activity*
choice	*Activity*
in	*Split*
free	*ControlPort*

Table 1: Advice types and joinpoints

Activity pointcuts
activity(identifierpattern)
compositeactivity(identifierpattern)
atomicactivity(identifierpattern)
Split pointcuts
split(identifierpattern)
andsplit(identifierpattern)
xorsplit(identifierpattern)
Control port pointcuts
controlport(identifier)
controlinputport(identifier)
controloutputport(identifier)

Table 2: Pointcut predicates

Pointcuts are expressions that resolve to a set of joinpoints, and are used to specify where in the base concern the connector should add its functionality. Because all *Activities*, *Splits* and *ControlPorts* have names that are unique among their siblings, every joinpoint can be uniquely identified by prepending the name of the *Activity*, *Split* or *ControlPort* with the names of their parents. For example, the `ControlOut` control port of the `ReturnObjections` activity in the `SoftwareDevelopment` base concern can be uniquely identified as `SoftwareDevelopment.ReturnObjections.ControlOut`. This allows specifying sets of joinpoints as identifier patterns. Pointcuts can be expressed using the predicates in Table 2. For example, if one wants to select all *Activities* in the `SoftwareDevelopment` base concern whose names end with `Phase`, one can use the expression `executingactivity("SoftwareDevelopment\..*Phase")`.

There are seven kinds of *InversionOfControlConnectors*, one for each of the advice types listed in Table 1.

BeforeConnectors, *AfterConnectors*, and *AroundConnectors* allow inserting a certain *Activity* before, after, or around each member of a set of *Activities* in another concern. These correspond to the classic *before*, *after*, and *around* advice types that are common in aspect-oriented research.

ParallelConnectors and *ChoiceConnectors* allow adding a parallel or alternative *Activity* to each member of a set of *Activities* in another concern. These are novel advice types that have not yet been considered in aspect-oriented research.

InConnectors allow adding an *Activity* as an extra branch to an existing *Split*. These are similar to

PADUS's *in* advice type [2].

FreeConnectors allow (AND- or XOR-) splitting a concern's control flow into another *Activity* at a certain control port, and joining the concern at another control port. These control ports are specified using two pointcuts: the *splitting* pointcut and the *joining* pointcut, respectively. The splitting pointcut specifies where the concern's control flow will be split into the advice activity, and the joining pointcut specifies where the concern will be joined. *FreeConnectors* are more general than *Parallel-*, *Choice-*, and *InConnectors*: *Parallel-* and *ChoiceConnectors* allow adding a parallel or alternative *Activity* to an existing *Activity* and *InConnectors* allow adding an *Activity* as an extra branch to an existing *Split*, whereas *FreeConnectors* allow more freedom in where the control flow of the base concern is split into the advice concern, and where the advice concern joins the control flow of the base concern.

The concrete syntax of the connectors is not relevant at this time: this technical report only deals with the connectors' semantics, which is discussed in the next sections.

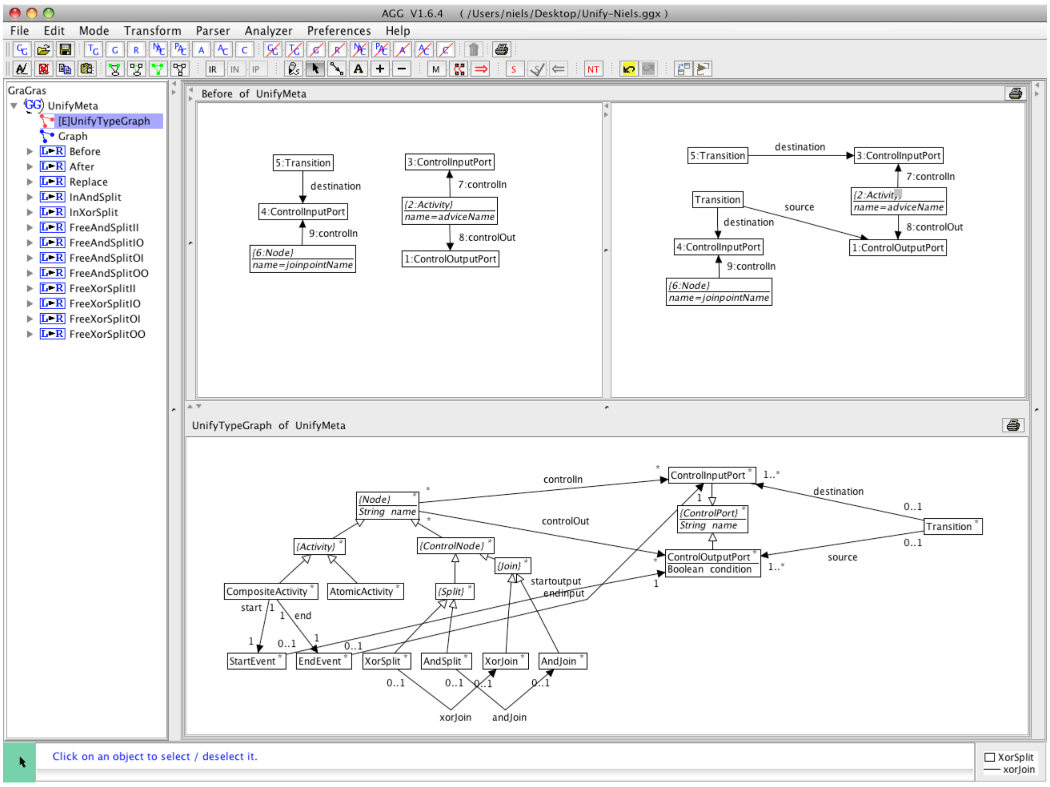
4 Graph Transformation

The semantics of a connector, which connects an advice concern to a base concern, is given by constructing a new concern that composes the base concern and the advice concern according to the connector type and the pointcut specification. This is accomplished using *graph transformation rules* that work on the abstract syntax of the UNIFY base language.

A *graph* consists of a set of nodes and a set of edges. A *typed graph* is a graph in which each node and edge belong to a type defined in a *type graph*. An *attributed graph* is a graph in which each node and edge may contain attributes where each attribute is a $(value, type)$ pair giving the value of the attribute and its type. Types can be structured by an inheritance relation.

A *graph transformation rule* is a rule used to modify a host graph, G, and is defined by two graphs (L, R). L is the left-hand side of the rule representing the pre-conditions of the rule and R is the right-hand side representing the post-conditions of the rule. The process of applying the rule to a graph G involves finding a graph monomorphism, h, from L to G and replacing $h(L)$ in G with $h(R)$. Further details can be found in [11].

In our approach, the type graph represents the meta-model shown in Figure 1. The translation of this meta-model to a type graph is straightforward: each meta-class corresponds to a typed node and each meta-association corresponds to a typed edge. Attributes in the meta-model are translated to corresponding node attributes. The wellformedness constraints can be formalized by graph constraints. Figure 3 shows a screenshot of UNIFY's type graph and Before rule in AGG [13].



Click on an object to select / deselect it.

Figure 3: Screenshot of UNIFY's type graph (bottom) and Before rule (top) in AGG

5 Rules

5.1 BeforeConnector

The rule for the *BeforeConnector* is parametrized by the name of a joinpoint *Activity*, and the name of the advice *Activity* that should be added before it. The evaluation of the regular expressions used in the pointcut predicates `executingactivity`, `executingcompositeactivity` and `executingatomicactivity` results in a set of joinpoint *Activity* names. Each name is the input for a rule application. Figure 4 shows the **Before(joinpointName : String, adviceName : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be augmented (i.e., an *Activity* whose name is the value of the `joinpointName` parameter, together with its *ControlInputPort* and the *Transition* that is connected to it) and the advice *Activity* named `adviceName` with its corresponding input and output ports. The right-hand side of the rule shows the connection of the original *Transition* to the advice *Activity*'s *ControlInputPort*, and of the advice *Activity*'s *ControlOutputPort* to the joinpoint *Activity*'s *ControlInputPort* through a new *Transition*.

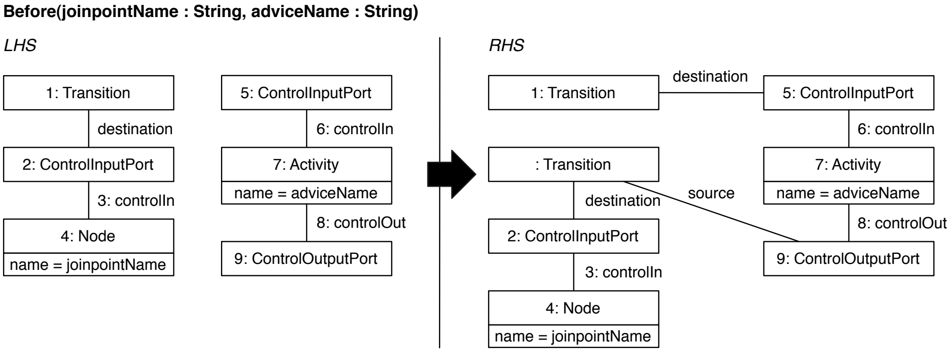


Figure 4: The `Before` rule

5.2 AfterConnector

The rule for the *AfterConnector* is similar to the rule for the *BeforeConnector*. It is parametrized by the name of a joinpoint *Activity*, and the name of the advice *Activity* that should be added after it. The evaluation of the regular expressions used in the pointcut predicates `executingactivity`, `executingcompositeactivity` and `executingatomicactivity` results in a set of joinpoint *Activity* names. Each name is the input for a rule application. Figure 5 shows the **After(joinpointName : String, adviceName : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be augmented (i.e., an *Activity* whose name is the value of the `joinpointName` parameter, together with its *ControlOutputPort* and the *Transition* that is connected to it) and the advice *Activity* named `adviceName` with its corresponding input and output ports. The right-hand side of the rule shows the connection of the joinpoint *Activity*'s *ControlOutputPort* to the advice *Activity*'s *ControlInputPort* through a new *Transition*, and of the advice *Activity*'s *ControlOutputPort* to the original *Transition*.

5.3 AroundConnector

The rule for the *AroundConnector* is parametrized by the name of a joinpoint *Activity*, the name of the advice *CompositeActivity* that should be woven around it, and the name of the proceed *Activity* (which is a child of the advice *CompositeActivity* and indicates where the joinpoint *Activity* should occur within the advice). The evaluation of the regular expressions used in the pointcut predicates

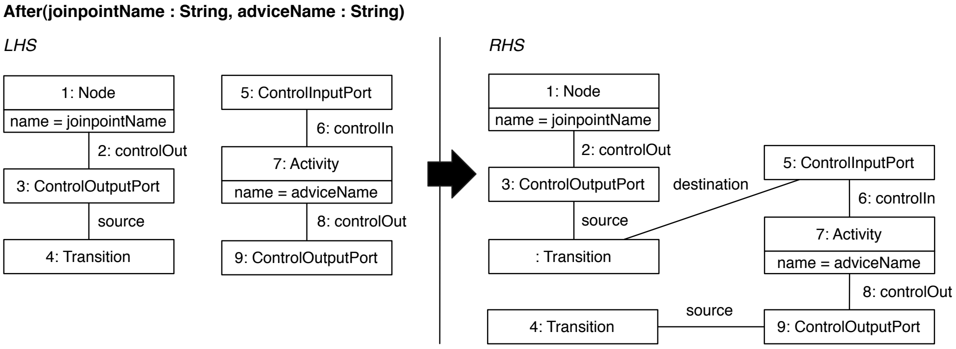


Figure 5: The `After` rule

`executingactivity`, `executingcompositeactivity` and `executingatomicactivity` results in a set of joinpoint *Activity* names. Each name is the input for a rule application. Figure 6 shows the **Around(joinpointName : String, adviceName : String, proceedName : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be augmented: an *Activity* whose name is the value of the `joinpointName` parameter (together with its *ControlInputPort* and *ControlOutputPort*, and the *Transitions* that are connected to them), the advice *CompositeActivity* named `adviceName` with its corresponding control input and output ports, and the advice *CompositeActivity*'s child *Activity* named *proceedName* (together with its *ControlInputPort* and *ControlOutputPort*, and the *Transitions* that are connected to them). The right-hand side of the rule shows the connection of the original incoming *Transition* to the advice *Activity*'s *ControlInputPort*, and of the advice *Activity*'s *ControlOutputPort* to the original outgoing *Transition*. As a child of the advice *Activity*, the proceed *Activity* is replaced by the joinpoint *Activity*.

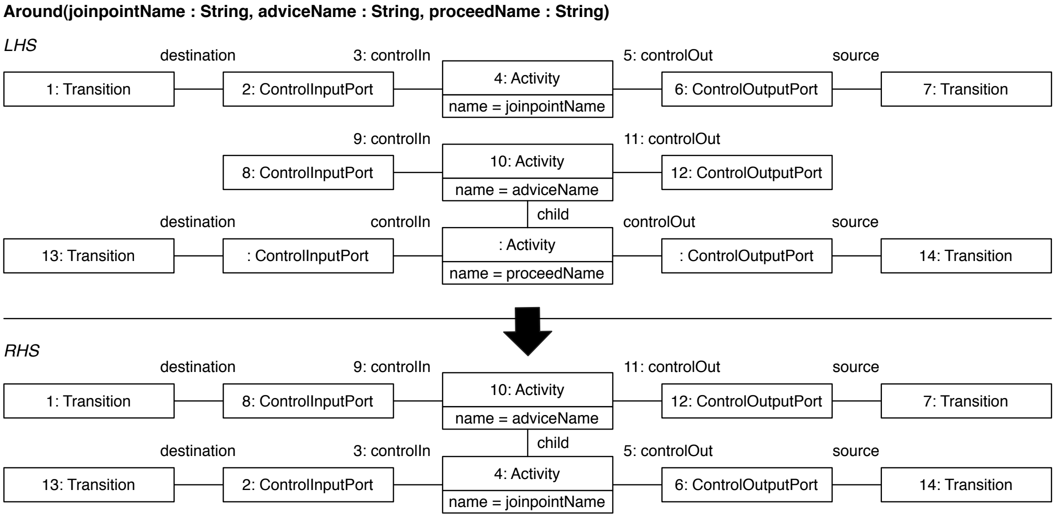


Figure 6: The `Around` rule

5.4 ParallelConnector

The rule for the *ParallelConnector* is parametrized by the name of a joinpoint *Activity*, and the name of the advice *Activity* that should be added parallel to it. The evaluation of the regular expressions used in the pointcut predicates `executingactivity`, `executingcompositeactivity` and `execut-`

`ingatomicactivity` results in a set of joinpoint *Activity* names. Each name is the input for a rule application. Figure 7 shows the **Parallel(joinpointName : String, adviceName : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be augmented (i.e., an *Activity* whose name is the value of the `joinpointName` parameter, together with its *ControlInputPort* and the incoming *Transition* that is connected to it, and its *ControlOutputPort* and the outgoing *Transition* that is connected to it) and the advice *Activity* named `adviceName` with its corresponding input and output ports. The right-hand side of the rule shows the connection of the original incoming *Transition* to a new *AndSplit* (through a new *ControlInputPort*). The new *AndSplit* has two outgoing branches: the first connects the new *AndSplit* to the joinpoint *Activity*'s *ControlInputPort* (through a new *ControlOutputPort* and *Transition*), while the second connects the new *AndSplit* to the advice *Activity*'s *ControlInputPort* (through a new *ControlOutputPort* and *Transition*). The joinpoint *Activity*'s *ControlOutputPort* is connected to a new *AndJoin* (through a new *Transition* and *ControlInputPort*), just like the advice *Activity*'s *ControlOutputPort* is connected to this new *AndJoin* (through a new *Transition* and *ControlInputPort*). Finally, the new *AndJoin* is connected to the joinpoint *Activity*'s original outgoing *Transition* (through a new *ControlOutputPort*).

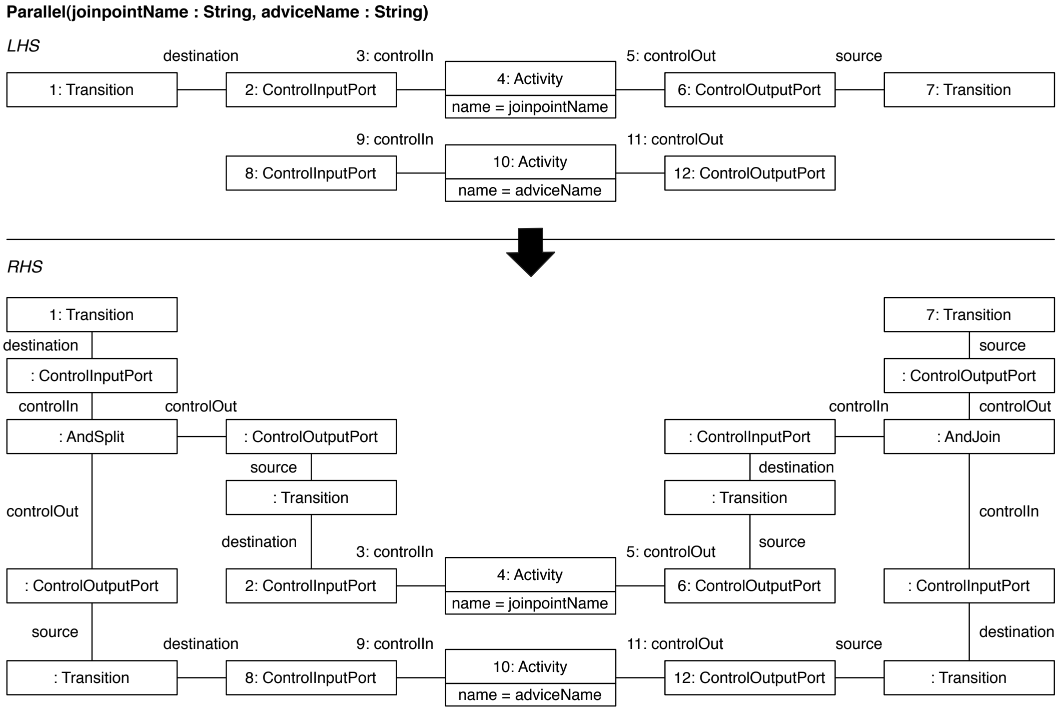


Figure 7: The `Parallel` rule

5.5 ChoiceConnector

The rule for the *ChoiceConnector* is similar to the rule for the *ParallelConnector*. It is parametrized by the name of a joinpoint *Activity*, the name of the advice *Activity* that should be added alternative to it, and the condition that decides whether the alternative branch should be followed or not. The evaluation of the regular expressions used in the pointcut predicates `executingactivity`, `executingcompositeactivity` and `executingatomicactivity` results in a set of joinpoint *Activity* names. Each name is the input for a rule application. Figure 8 shows the **Choice(joinpointName : String, adviceName : String, sCondition : String)** rule. The left-hand side of the rule specifies the partial match of the workflow that will be augmented (i.e., an *Activity* whose name is the value of the `joinpointName` parameter, together with its *ControlInputPort* and the incoming *Transition* that is connected to it, and its *ControlOutputPort* and the outgoing *Transition* that is connected to it) and the advice *Activity* named `adviceName` with its corresponding input and output ports. The right-hand side of the rule shows the connection of the original incoming *Transition* to a new *XorSplit* (through a new *ControlInputPort*). The new *XorSplit* has two outgoing branches: the first connects the new *AndSplit* to the joinpoint *Activity*'s *ControlInputPort* (through a new *ControlOutputPort* and *Transition*), while the second connects the new *AndSplit* to the advice *Activity*'s *ControlInputPort* (through a new *ControlOutputPort* and *Transition*) using the specified condition. The joinpoint *Activity*'s *ControlOutputPort* is connected to a new *XorJoin* (through a new *Transition* and *ControlInputPort*), just like the advice *Activity*'s *ControlOutputPort* is connected to this new *XorJoin* (through a new *Transition* and *ControlInputPort*). Finally, the new *XorJoin* is connected to the joinpoint *Activity*'s original outgoing *Transition* (through a new *ControlOutputPort*).

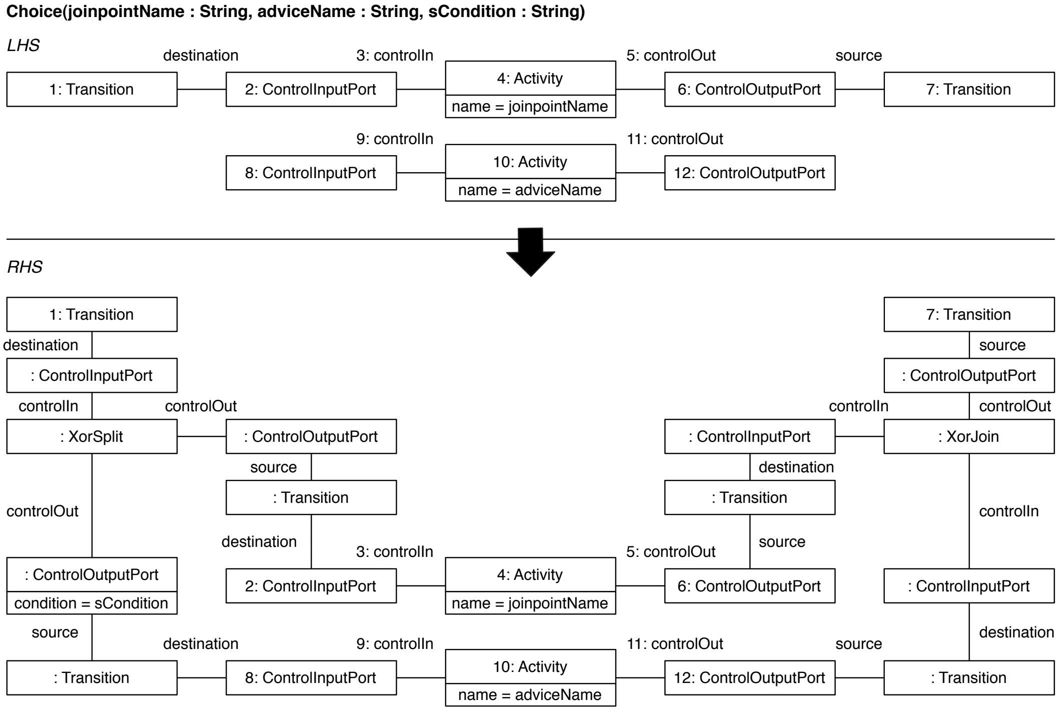


Figure 8: The `Choice` rule

5.6 InConnector

Each rule for the *InConnector* is parametrized by the name of a *Split* and the name of the *Activity* that is to be added as an extra branch to this split. The evaluation of the regular expressions used in the pointcut predicates `split`, `andsplit` and `xorsplit` results in a set of *Split* names. Each name is the input for a rule application. Figure 9 shows the **InAndSplit(sName : String, aName : String)** rule corresponding to an *InConnector* with pointcut predicate `andsplit`. The left-hand side of the rule specifies the partial match of the workflow that will be augmented (i.e., an *AndSplit* whose name is the value of the `sName` parameter) and the advice *Activity* named `aName` which can be an *AtomicActivity* or a *CompositeActivity* with its corresponding input and output ports. The right-hand side of the rule shows the addition of a branch to the *AndSplit* by adding a *ControlOutputPort* to it, and connecting it to the *ControlInputPort* of the advice *Activity* using a new *Transition*. A *ControlInputPort* is added to the *AndSplit*'s corresponding *AndJoin*, and is connected to the advice *Activity*'s *ControlOutputPort* using a new *Transition*. Remark that in the left-hand side of the rule we demand the existence of a corresponding *AndJoin* for the *AndSplit*. This means that the workflow developer needs to take care of adding a join for each split or we assume a preprocessing step where corresponding joins to splits are inserted in the workflow if possible.

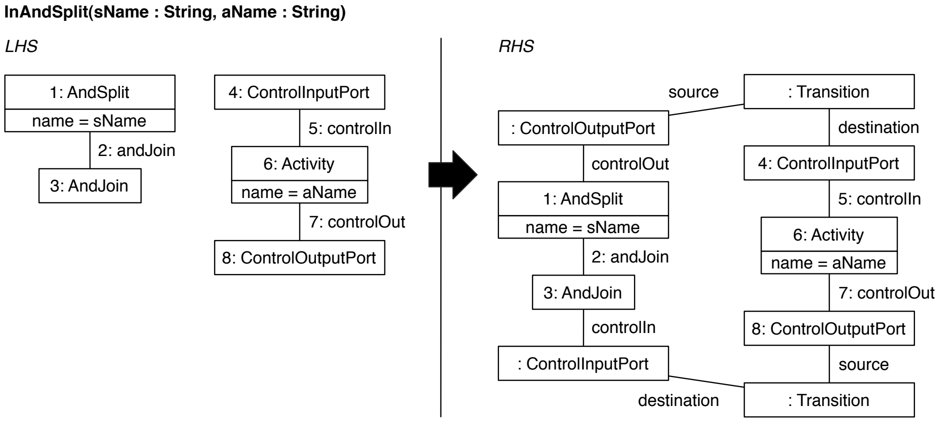


Figure 9: The `InAndSplit` rule

The composition rule for the *InConnector* and the `xorsplit` pointcut predicate is similar, and is given in Figure 10 by **InXorSplit(sName : String, aName : String)**. A rule for the *InConnector* and the `split` pointcut predicate is not necessary because each split identified by the evaluation of the regular expression in the `split` predicate is an *AndSplit* or an *XorSplit*. As a result of this either the `InAndSplit` or the `InXorSplit` rule will be triggered.

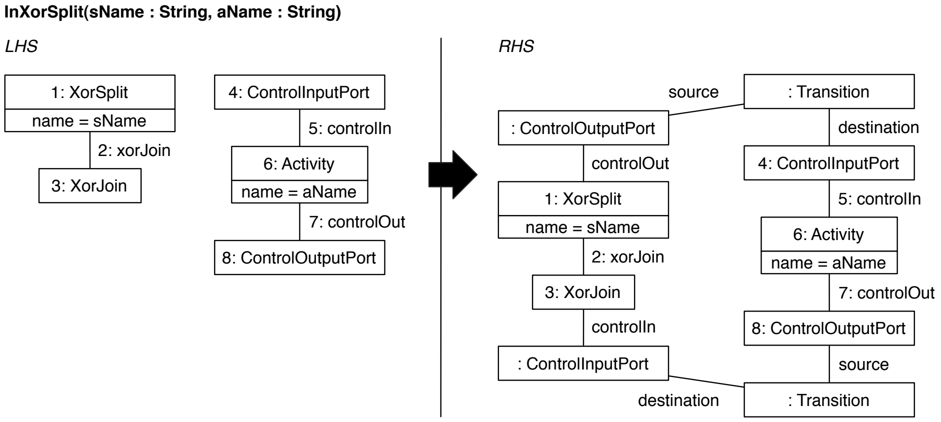


Figure 10: The `InXorSplit` rule

5.7 FreeConnector

A *FreeConnector* can vary in its splitting behavior, which is either AND-splitting or XOR-splitting, and in the types of its splitting and joining control ports, which are either input/input, input/output, output/input, or output/output. Thus, there are eight rules in total, as shown in Table 3.

The **FreeAndSplitII(sName : String, jName : String, aName : String)** rule, which is given in Figure 11, adds a split at a certain control input port and joins at another control input port. The rule is parametrized with the name of the splitting control input port, the name of the joining control input port, and the name of the activity that is to be inserted. The left-hand side of the rule specifies the splitting *ControlInputPort* and its incoming *Transition*, the joining *ControlInputPort* and its

Splitting behavior	Splitting control port	Joining control port	Rule name	Figure
AND-splitting	input	input	FreeAndSplitII	Figure 11
		output	FreeAndSplitIO	Figure 13
	output	input	FreeAndSplitOI	Figure 15
		output	FreeAndSplitOO	Figure 17
XOR-splitting	input	input	FreeXorSplitII	Figure 12
		output	FreeXorSplitIO	Figure 14
	output	input	FreeXorSplitOI	Figure 16
		output	FreeXorSplitOO	Figure 18

Table 3: The FreeConnector rules

incoming *Transition*, and the advice *Activity* with its *ControlInputPort* and *ControlOutputPort*. The right-hand side specifies the graph after inserting the free advice. The splitting *ControlInputPort*'s original incoming *Transition* is now connected to a new *AndSplit*. The *AndSplit* has two outgoing *Transitions*, the first is a new *Transition* towards the splitting *ControlInputPort*, and the second is a new *Transition* towards the *ControlInputPort* of the advice *Activity*. The *ControlOutputPort* of the advice *Activity* is connected to a new *AndJoin* through a new *Transition*. The other incoming *Transition* of the *AndJoin* is the joining *ControlInputPort*'s original incoming *Transition*. Finally, the *AndJoin*'s outgoing *Transition* is a new *Transition* towards the joining *ControlInputPort*.

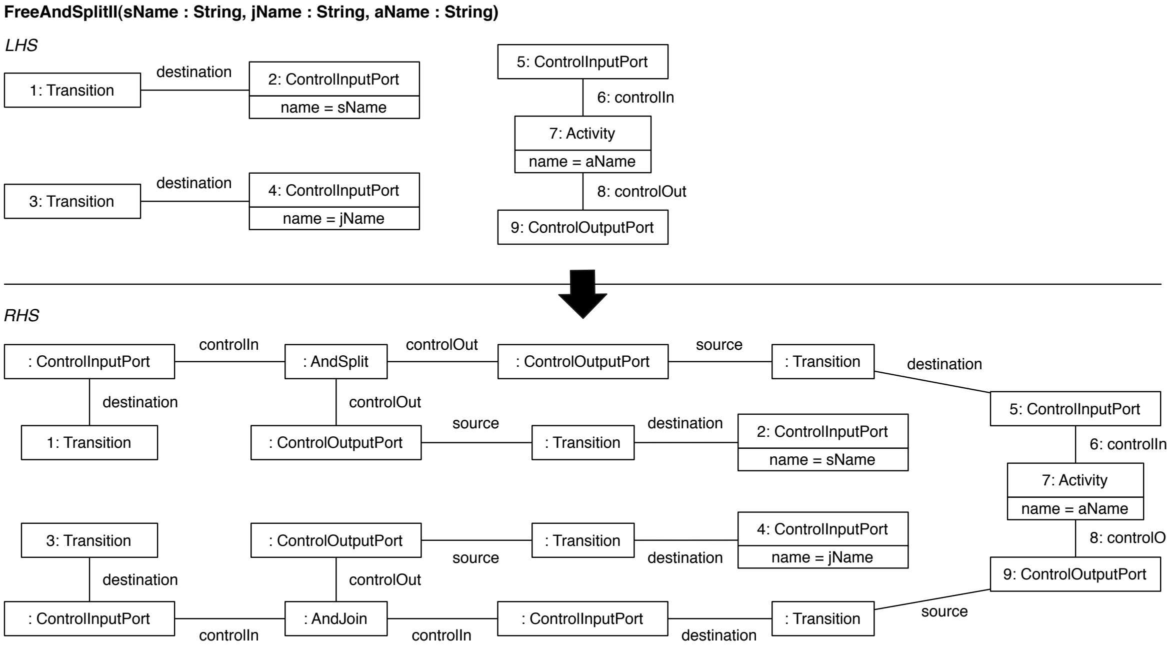


Figure 11: The `FreeAndSplitII` rule

The **FreeXorSplitII(sName : String, jName : String, aName : String, sCondition : String)** rule, which is given in Figure 12, is analogous: the only difference is that it inserts an *XorSplit* (with the appropriate splitting condition) and an *XorJoin* instead of an *AndSplit* and an *AndJoin*.

The **FreeAndSplitIO(sName : String, jName : String, aName : String)** rule, which is given in Figure 13, adds a split at a certain control input port and joins at a certain control output port. The rule is parametrized with the name of the splitting control input port, the name of the joining control output port, and the name of the activity that is to be inserted. The left-hand side of the rule specifies the splitting *ControlInputPort* and its incoming *Transition*, the joining *ControlOutputPort* and its

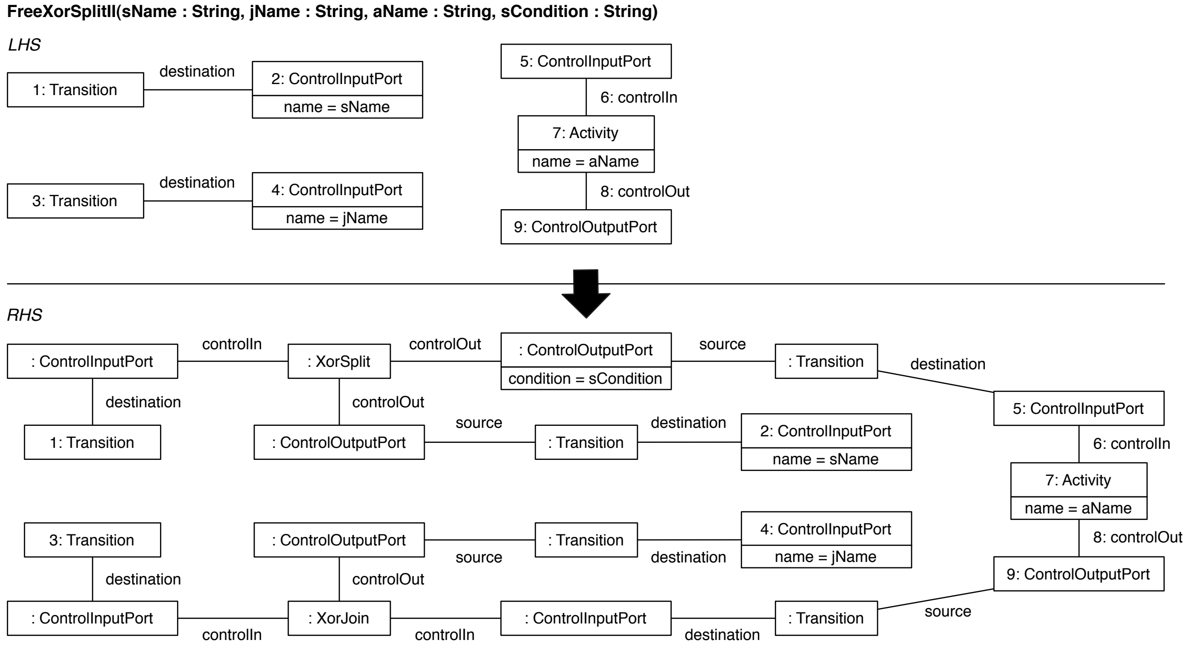


Figure 12: The `FreeXorSplitII` rule

outgoing *Transition*, and the advice *Activity* with its *ControlInputPort* and *ControlOutputPort*. The right-hand side specifies the graph after inserting the free advice. The splitting *ControlInputPort*'s original incoming *Transition* is now connected to a new *AndSplit*. The *AndSplit* has two outgoing *Transitions*, the first is a new *Transition* towards the splitting *ControlInputPort*, and the second is a new *Transition* towards the *ControlInputPort* of the advice *Activity*. The *ControlOutputPort* of the advice *Activity* is connected to a new *AndJoin* through a new *Transition*. The other incoming *Transition* of the *AndJoin* is a new *Transition* that comes from the joining *ControlOutputPort*. Finally, the *AndJoin*'s outgoing *Transition* is the joining *ControlOutputPort*'s original outgoing *Transition*.

The **FreeXorSplitIO(sName : String, jName : String, aName : String, sCondition : String)** rule, which is given in Figure 14, is analogous: the only difference is that it inserts an *XorSplit* (with the appropriate splitting condition) and an *XorJoin* instead of an *AndSplit* and an *AndJoin*.

The **FreeAndSplitOI(sName : String, jName : String, aName : String)** rule, which is given in Figure 15, adds a split at a certain control output port and joins at a certain control input port. The rule is parametrized with the name of the splitting control output port, the name of the joining control input port, and the name of the activity that is to be inserted. The left-hand side of the rule specifies the splitting *ControlOutputPort* and its outgoing *Transition*, the joining *ControlInputPort* and its incoming *Transition*, and the advice *Activity* with its *ControlInputPort* and *ControlOutputPort*. The right-hand side specifies the graph after inserting the free advice. The splitting *ControlOutputPort* is now connected to a new *AndSplit* through a new *Transition*. The *AndSplit* has two outgoing *Transitions*, the first is the splitting *ControlOutputPort*'s original outgoing *Transition*, and the second is a new *Transition* towards the *ControlInputPort* of the advice *Activity*. The *ControlOutputPort* of the advice *Activity* is connected to a new *AndJoin* through a new *Transition*. The other incoming *Transition* of the *AndJoin* is the joining *ControlInputPort*'s original incoming *Transition*. Finally, the *AndJoin*'s outgoing *Transition* is a new *Transition* towards the joining *ControlInputPort*.

The **FreeXorSplitOI(sName : String, jName : String, aName : String, sCondition : String)** rule, which is given in Figure 16, is analogous: the only difference is that it inserts an *XorSplit* (with the appropriate splitting condition) and an *XorJoin* instead of an *AndSplit* and an *AndJoin*.

The **FreeAndSplitOO(sName : String, jName : String, aName : String)** rule, which is given in Fig-

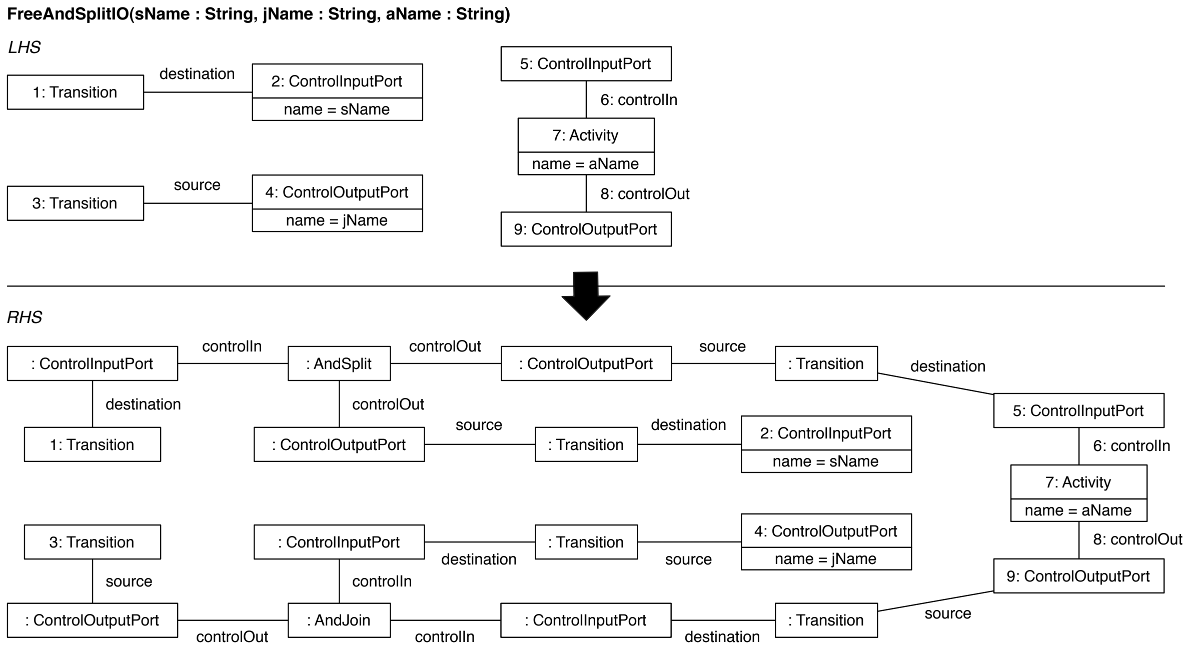


Figure 13: The `FreeAndSplitIO` rule

ure 17, adds a split at a certain control output port and joins at another control output port. The rule is parametrized with the name of the splitting control output port, the name of the joining control output port, and the name of the activity that is to be inserted. The left-hand side of the rule specifies the splitting *ControlOutputPort* and its outgoing *Transition*, the joining *ControlOutputPort* and its outgoing *Transition*, and the advice *Activity* with its *ControlInputPort* and *ControlOutputPort*. The right-hand side specifies the graph after inserting the free advice. The splitting *ControlOutputPort* is now connected to a new *AndSplit* through a new *Transition*. The *AndSplit* has two outgoing *Transitions*, the first is the splitting *ControlOutputPort*'s original outgoing *Transition*, and the second is a new *Transition* towards the *ControlInputPort* of the advice *Activity*. The *ControlOutputPort* of the advice *Activity* is connected to a new *AndJoin* through a new *Transition*. The other incoming *Transition* of the *AndJoin* is a new *Transition* that comes from the joining *ControlOutputPort*. Finally, the *AndJoin*'s outgoing *Transition* is the joining *ControlOutputPort*'s original outgoing *Transition*.

The **FreeXorSplitOO(sName : String, jName : String, aName : String, sCondition : String)** rule, which is given in Figure 18, is analogous: the only difference is that it inserts an *XorSplit* (with the appropriate splitting condition) and an *XorJoin* instead of an *AndSplit* and an *AndJoin*.

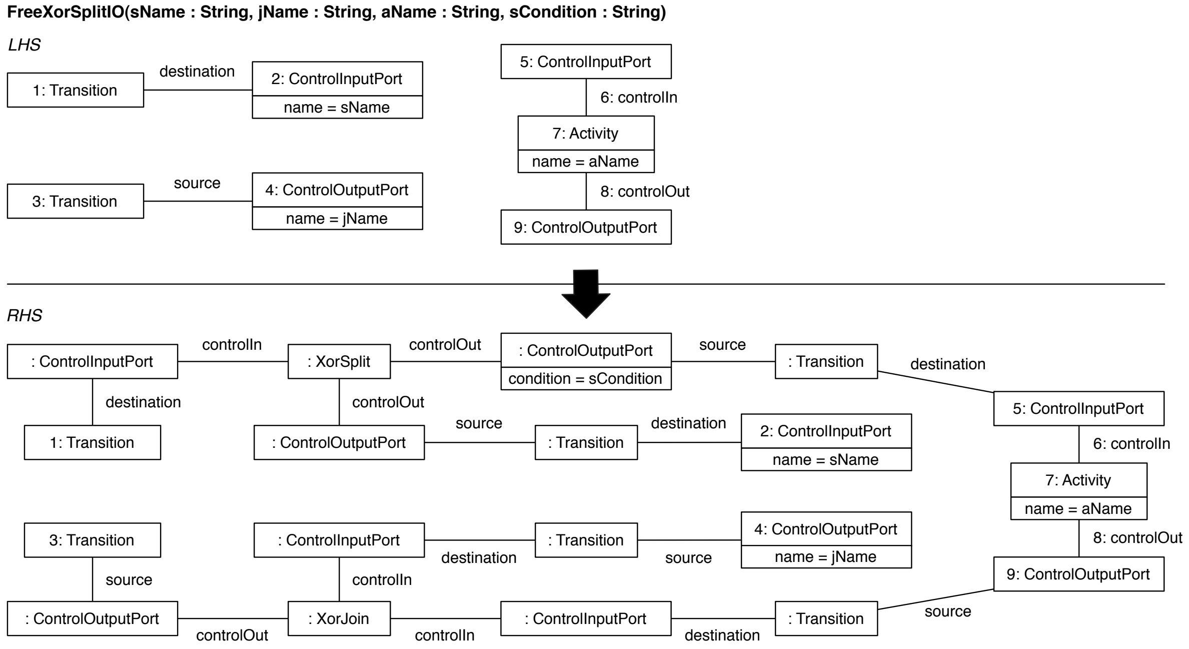


Figure 14: The `FreeXorSplitIO` rule

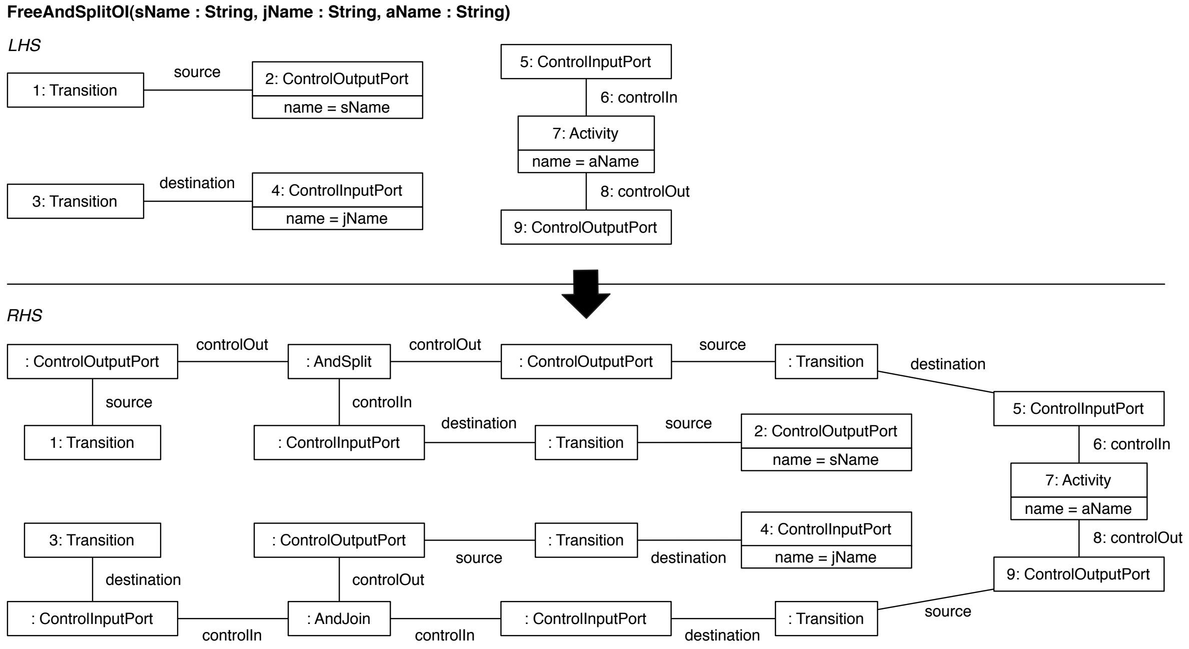


Figure 15: The `FreeAndSplitOI` rule

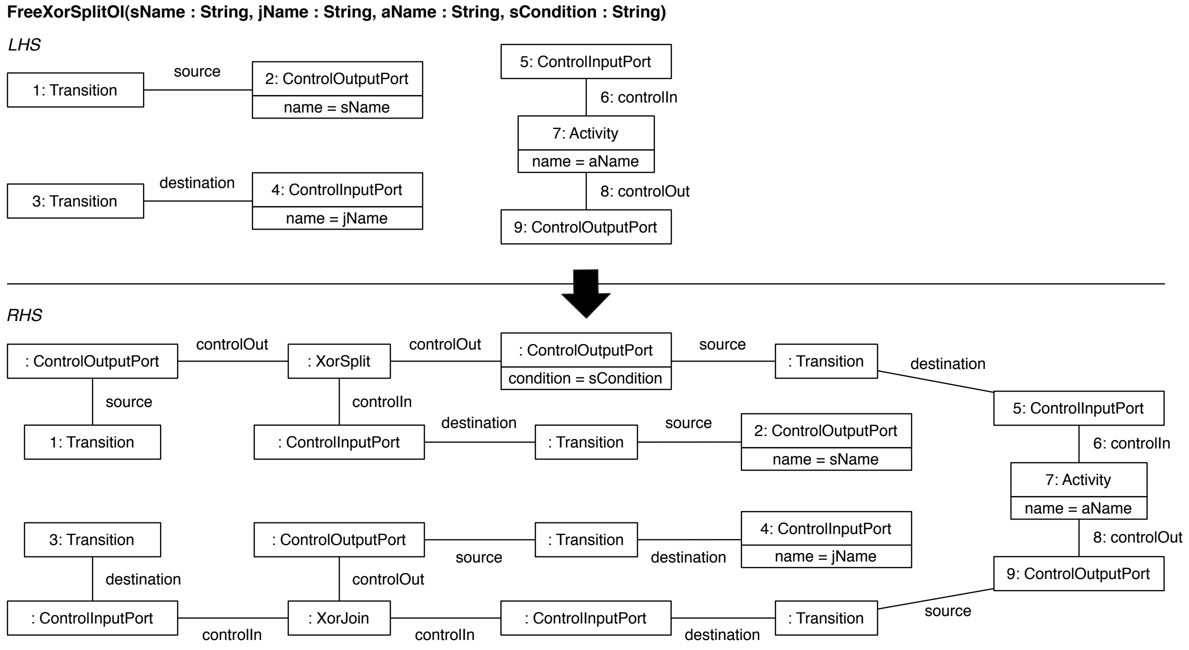


Figure 16: The FreeXorSplitOI rule

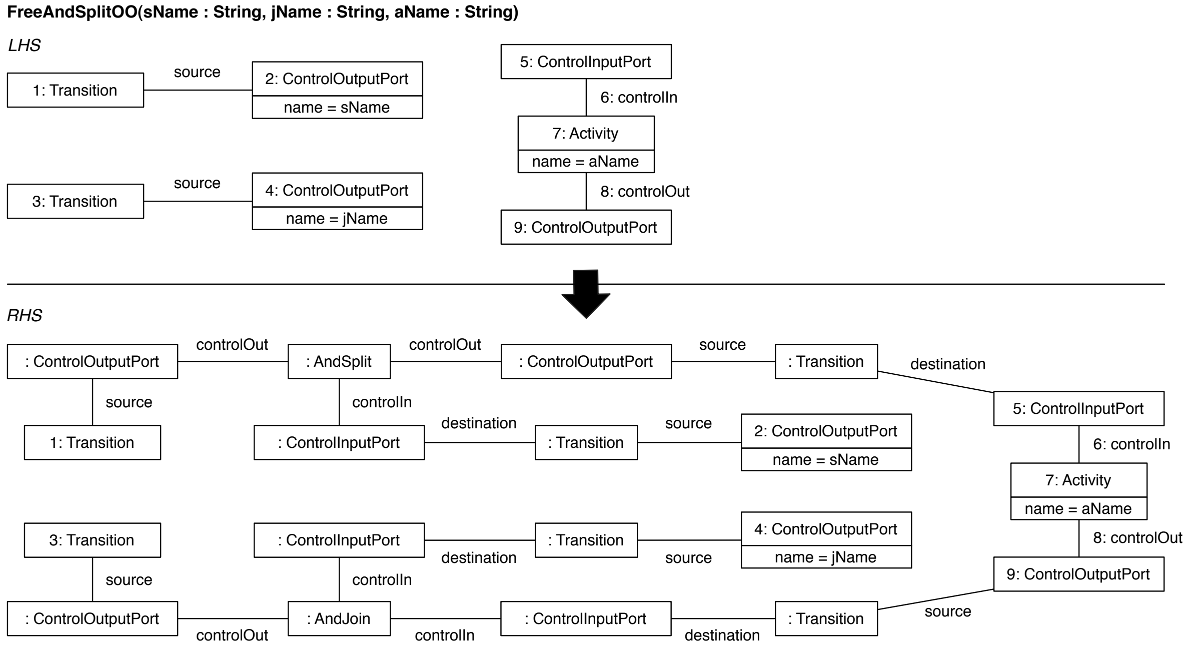


Figure 17: The FreeAndSplitOO rule

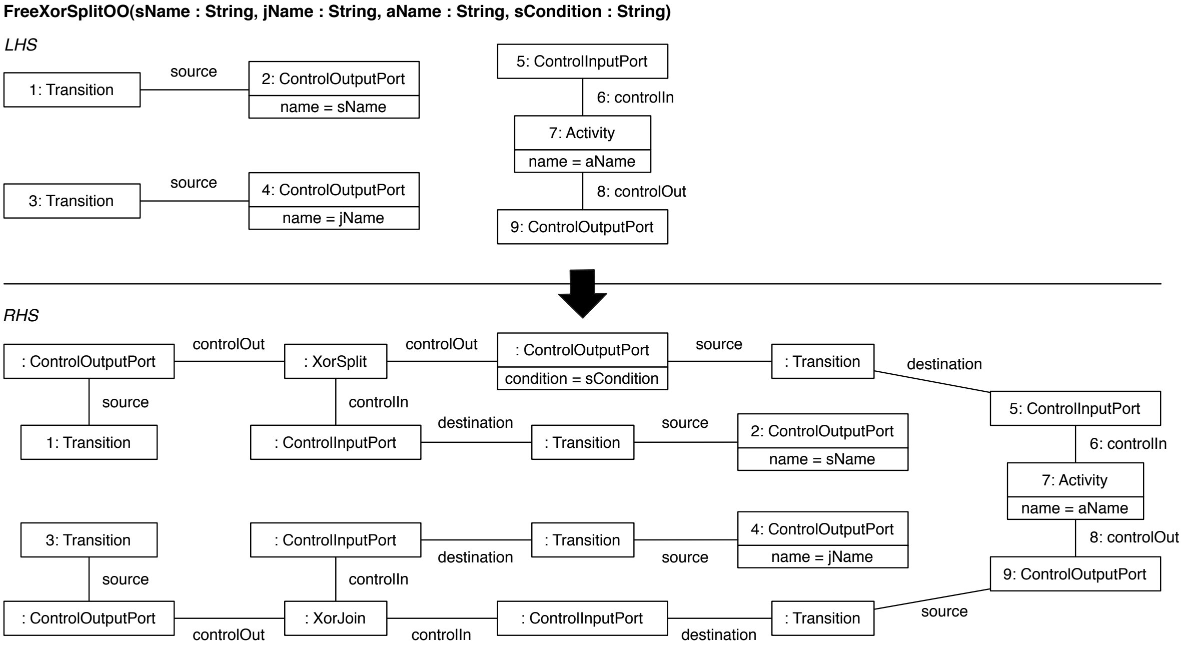


Figure 18: The `FreeXorSplitOO` rule

6 Conclusions

Most state-of-the-art workflow languages offer a limited set of modularization mechanisms. This typically results in monolithic workflow specifications, in which different concerns are scattered across the workflow and tangled with each other. This hinders the design, the evolution, and the reusability of workflows expressed in these languages.

We address this problem by introducing the UNIFY framework, which supports advanced modularization of workflows based on aspect-oriented principles. UNIFY allows specifying each concern in isolation of other concerns, and provides a connector mechanism that allows connecting these concerns according to workflow-specific connection patterns. This technical report gives a detailed description of the semantics of the connector construct by providing a graph transformation rule for each combination of connector type and pointcut predicate.

References

[1] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services, version 1.1, May 2003. 3

[2] Mathieu Braem, Kris Verlaenen, Niels Joncheere, Wim Vanderperren, Ragnhild Van Der Straeten, Eddy Truyen, Wouter Joosen, and Viviane Jonckers. Isolating process-level concerns using Padus. In *Proceedings of the 4th International Conference on Business Process Management (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 113–128, Vienna, Austria, September 2006. Springer. 3, 7, 8

[3] Anis Charfi and Mira Mezini. Aspect-oriented web service composition with AO4BPEL. In *Proceedings of the 2nd European Conference on Web Services (ECOWS 2004)*, volume 3250 of *Lecture Notes in Computer Science*, pages 168–182, Erfurt, Germany, September 2004. Springer. 3, 7

[4] Carine Courbis and Anthony Finkelstein. Towards aspect weaving applications. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 69–77, St. Louis, MO, USA, May 2005. ACM Press. 3, 7

[5] Oscar Gonzalez, Rubby Casallas, and Dirk Deridder. MMC-BPM: A domain-specific language for business process analysis. In *Proceedings of the 12th International Conference on Business Information Systems (BIS 2009)*, volume 21 of *Lecture Notes in Business Information Processing*, pages 157–168, Poznań, Poland, April 2009. Springer. 3

[6] Niels Joncheere, Dirk Deridder, Ragnhild Van Der Straeten, and Viviane Jonckers. A framework for advanced modularization and data flow in workflow systems. In *Proceedings of the 6th International Conference on Service Oriented Computing (ICSOC 2008)*, volume 5364 of *Lecture Notes in Computer Science*, pages 592–598, Sydney, NSW, Australia, December 2008. Springer. 3

[7] Niels Joncheere and Ragnhild Van Der Straeten. Uniform modularization of workflow concerns using Unify. In *Proceedings of the 4th International Conference on Software Language Engineering (SLE 2011)*. (to appear). 3

[8] Diane Jordan, John Evdemon, et al. Web Services Business Process Execution Language, version 2.0, April 2007. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`. 6

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer. 3

[10] Object Management Group. Business Process Model and Notation, version 2.0, January 2011. `http://www.omg.org/spec/BPMN/2.0/`. 6

[11] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, River Edge, NJ, USA, 1997. 9

[12] Nick Russell, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, and Nataliya Mulyar. Workflow control-flow patterns: A revised view. BPM Center Report BPM-06-22, BPM Center, 2006. 6

[13] Gabriele Taentzer et al. The Attributed Graph Grammar system: A development environment for attributed graph transformation systems. `http://user.cs.tu-berlin.de/~gragra/agg/`. 9

[14] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 107–119, Los Angeles, CA, USA, May 1999. IEEE Computer Society. 3

VRIJE UNIVERSITEIT BRUSSEL • SCIENTIA VINCERE TENEBRAS •

Vrije Universiteit Brussel - Software Languages Lab
Pleinlaan 2, 1050 Brussels, Belgium - http://soft.vub.ac.be/