Uniform Modularization of Workflow Concerns using Unify

Niels Joncheere* and Ragnhild Van Der Straeten

Vrije Universiteit Brussel
Software Languages Lab
Pleinlaan 2, 1050 Brussels, Belgium
{njonchee,rvdstrae}@vub.ac.be

Abstract. 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 one another. 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 uniform modularization of workflows by allowing all workflow concerns — including crosscutting ones — to be specified in isolation of each other. These independently specified workflow concerns can then be connected to each other using a number of workflow-specific connectors. We discuss the interaction of the most invasive connector with the workflows' control flow and data perspectives. We instantiate the framework towards two state-of-the-art workflow languages, i.e., WS-BPEL and BPMN.

1 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. A workflow is created by dividing a process into different activities, and by specifying the ordering in which these activities need to be performed. This ordering is called the control flow perspective.

Separation of concerns [1] is a general software engineering principle that refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose. These parts, called *concerns*, are the primary motivation for organizing and decomposing software into manageable and comprehensible modules.

Realistic workflows consist of several concerns, which are connected in order to achieve the desired behavior. However, if all of these concerns need to be specified in a single, monolithic workflow specification, it will be hard to add,

* Funded by the Belgian State – Belgian Science Policy through the Interuniversity Attraction Poles program.

maintain, remove or reuse these concerns. Although most workflow languages allow decomposing workflows into sub-workflows, this mechanism is typically aimed at grouping activities instead of facilitating the independent evolution and reuse of concerns. Moreover, a workflow can only be decomposed according to one dimension with this construct, and concerns that do not align with this decomposition end up scattered across the workflow and tangled with one another. Such concerns are called *crosscutting concerns* [2]. These problems have been discussed in related work by ourselves [3, 4] and others [5–7], where they are mainly tackled using *aspect-oriented programming* for workflows. Nevertheless, the problems are not yet fully addressed by the proposed solutions.

The goal of our current solution is to facilitate independent evolution and reuse of *all* workflow concerns, i.e., not merely crosscutting concerns. This can be accomplished by improving the modularization mechanisms offered by the workflow language. We propose an approach called Unify which provides a set of workflow-specific modularization mechanisms that can be readily employed by a wide range of existing workflow languages. Unify facilitates specifying workflow concerns as separate modules. These modules are then composed using versatile connectors, which specify how the concerns are connected. The main contributions of Unify are the following:

1. Existing research on modularization of workflow concerns is aimed at only modularizing crosscutting concerns [6, 7, 3], or at only modularizing one particular kind of concern, such as monitoring [8]. 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 concern connection patterns that are not supported by other approaches.
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, extensible base language meta-model.
4. Unify defines a clear semantics for its modularization mechanism. 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 pre-processor that is compatible with existing workflow engines.

The structure of this paper is as follows. Section 2 specifies the motivation for Unify, and introduces a running example. Section 3 provides an initial description of our approach by showing how the running example could be developed from scratch when using Unify. Sections 4 and 5 introduce the meta-model for our base language and connector mechanism, respectively. Section 6 discusses the interaction of our connector mechanism with the control flow and data perspectives, and discusses the semantics of our connector mechanism. Section 7 describes our implementation, Section 8 gives an overview of related work, and Section 9 states our conclusions and outlines future work.

2 Motivation

Consider the workflow in Figure 1, which is a simplified version of an automated order handling process for an online book store. The workflow is visualized using the Business Process Model and Notation (BPMN) [9]; a brief overview of this notation is given in the legend at the bottom right of the figure. The workflow starts at the start event at the top of the figure. It first performs the *Login* and *SelectBooks* activities in parallel. The workflow then proceeds with the *SpecifyOptions* activity, after which the control flow is split again. A first branch contains the *Pay* and *SendInvoice* activities, while a second branch contains the *ProcessOrder* and *Ship* activities. The *VerifyBankAccount* activity synchronizes both branches. The last activity to be executed is the *ProcessReturns* activity, after which the workflow ends at the end event. Please note that only the contents of the *SelectBooks*, *Pay*, and *Ship* activities are shown, whereas the contents of other activities are omitted in the interest of brevity.

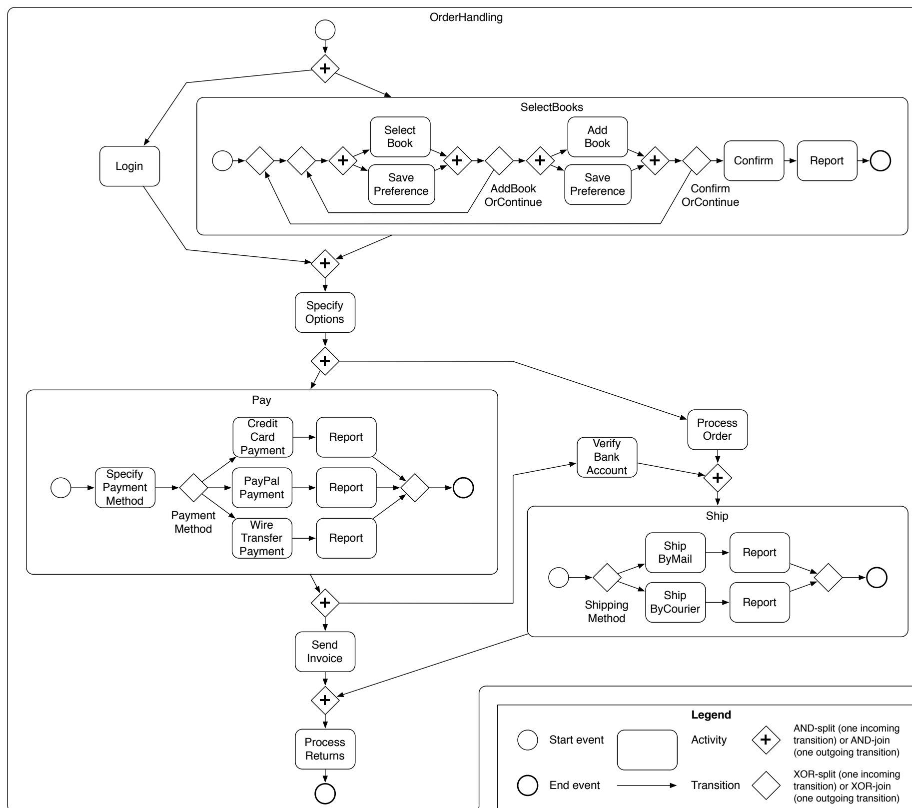

Fig. 1. Example order handling workflow, expressed using BPMN

Like any realistic software application, the workflow in Figure 1 consists of several *concerns* — parts that are relevant to a particular concept, goal, or purpose — which are connected in order to achieve the workflow's desired behavior. The main concern is obviously *order handling*. This concern has already been hierarchically decomposed into sub-concerns — such as *book selection*, *payment* and *shipping* — using the *composite activity* construct. Other concerns are *preference saving*, *reporting* and *bank account verification*, which occur at various places across the workflow. The general software engineering principle of *separation of concerns* argues that applications should be decomposed into different modules in such a way that each concern can be manipulated in isolation of other concerns. However, many current workflow languages do not allow decomposing workflows into different modules. For example, a workflow expressed using WS-BPEL [10] (the de facto standard in workflow languages) is a single, monolithic XML file that cannot be straightforwardly divided into sub-workflows. This lack of modularization mechanisms makes it hard to add, maintain, remove, or reuse concerns. In order to improve separation of concerns in workflows, workflow languages should allow concerns to be specified in isolation of each other.

However, allowing concerns to be specified in isolation of each other is not sufficient: in order to obtain the desired workflow behavior, workflow languages should also provide a means of specifying how a workflow's concerns are connected to each other.

In existing workflow languages, the only kind of connection that is supported is typically the classic sub-workflow pattern: a main workflow explicitly specifies that a sub-workflow should be executed. The choice of which sub-workflow is to be executed is made at design time, and it is hard to make a different choice afterwards. By delaying the choice of which sub-workflow is to be executed, the coupling between main workflow and sub-workflow is lowered, and separation of concerns is improved. In the workflow in Figure 1, one could for example vary the behavior of the workflow by deploying a different *Pay* sub-workflow in different situations.

A second kind of connection between concerns is useful when concerns *crosscut* a workflow: some concerns cannot be modularized cleanly using the sub-workflow decomposition mechanism, because they are applicable at several locations in the workflow. The reporting concern, for example, is present at several locations in the workflow in Figure 1. The sub-workflow construct does not solve this problem, since sub-workflows are called explicitly from within the main workflow. This makes it hard to add, maintain, remove or reuse such crosscutting concerns. This problem has been observed in general aspect-oriented research [2]. Aspect-oriented extensions to WS-BPEL, such as AO4BPEL [6] and Padus [3], allow specifying crosscutting concerns in separate aspects. An aspect allows specifying that a certain workflow fragment, called an *advice*, should be executed *before*, *after*, or *around* a certain set of activities in the base workflow. In the workflow in Figure 1, one could for example specify that the *Report* activity needs to be performed after the *Confirm* activity and after each of the three *Payment* and two *Ship* activities, without explicitly invoking the *Report*

activity at each of those places. However, these aspect-oriented extensions use a new language construct for specifying crosscutting concerns, i.e., aspects. This means that concerns which are specified using the aspect construct can only be reused as an aspect, and not as a sub-workflow. On the other hand, concerns which are specified using the sub-workflow construct can only be reused as a sub-workflow, and not as an aspect.

Moreover, the aspect-oriented extensions mentioned above only support the basic concern connection patterns (*before*, *after*, or *around*) that were identified in general aspect-oriented research, and do not sufficiently consider the specifics of the workflow context. They lack support for other patterns such as parallelism and choice. For example, the before, after or around patterns do not provide an elegant way of specifying that the *SavePreference* activity should be performed *in parallel with* the *SelectBook* and *AddBook* activities. Furthermore, it is completely impossible to specify more advanced connections between concerns, e.g., specifying that the *VerifyBankAccount* activity should be executed after the *Pay* activity has been executed and before the *Ship* activity is executed, which would thus synchronize the two parallel branches by introducing a new AND-split and -join in the order handling workflow.

Finally, the aspect-oriented extensions mentioned above are all targeted at WS-BPEL, and cannot be applied easily to other languages. Each of these approaches also favors a specific implementation technique; for example, AO4BPEL can only be executed using a modified WS-BPEL engine, and Padus can only be used as a pre-processor. More variability in terms of the applicable languages and possible implementation techniques would make a modularization approach more widely applicable.

3 Developing a Workflow using Unify

There are two main scenarios for applying Unify to workflow development. In the first scenario, Unify is used to improve a workflow that has already been developed using an existing workflow language, but without any regard for separation of concerns. Unify could then be used to decompose the existing workflow into a number of different modules, which each correspond to a concern, and which are connected to each other in order to achieve the original behavior. We will not consider this first scenario in this paper. The second scenario assumes that a developer is creating a new workflow from scratch, perhaps with a library of previously implemented concerns at his disposal. In this section, we will introduce Unify using this second scenario.

The first step in developing a workflow using Unify is to identify its concerns. In the example from Figure 1, these are, among others, *order handling*, *book selection*, *payment*, *shipping*, *preference saving*, and *reporting*. Unify promotes implementing a workflow's concerns as separate modules.[1] This can be achieved using the *composite activity* construct. Figure 2 shows how the concerns that

[1] Deciding which concerns should be modularized is partly a matter of personal preference, and is not the focus of our research.

were mentioned in the previous section could be specified separately. Note that each of the composite activities in Figure 2 contains less activities than the corresponding composite activity in Figure 1. The Unify base language, which is discussed in Section 4, defines the abstract syntax of our workflow concerns.

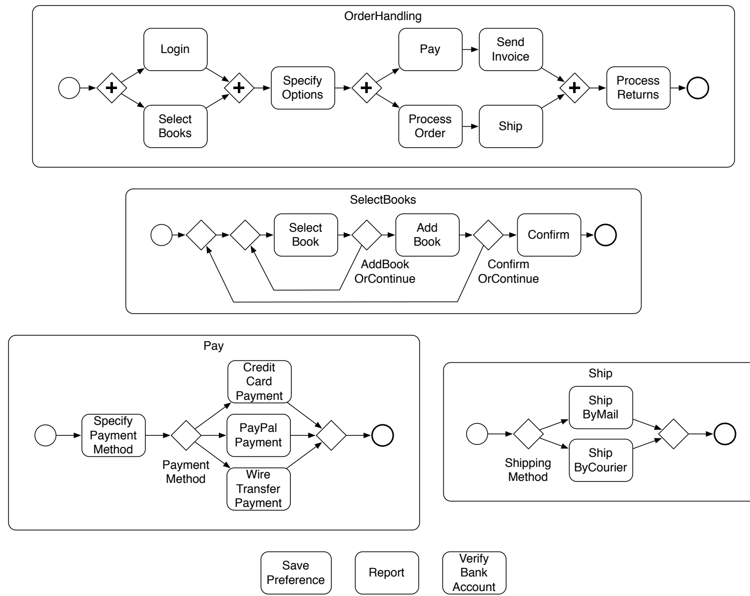

Fig. 2. Independently specified workflow concerns

The advantage of specifying workflow concerns as separate composite activities is better separation of concerns: the different parts of a concern are no longer scattered across the workflow(s), or tangled with one another. After the concerns have been identified and implemented (or retrieved from a library of previously implemented concerns), the connections between the concerns should be specified. We identify two main categories of connections between concerns:

– **Anticipated concern connections** are concern connections that are explicitly anticipated by one of the concerns: this concern is aware, at design time, of the fact that it will connect to another concern at a certain point in its execution.
– **Unanticipated concern connections** are concern connections that are *not* explicitly anticipated by the concerns: the concerns are not aware of the fact that they will connect to each other at a certain point in their execution.

An example of the former is apparent in the *OrderHandling* concern in Figure 2: this concern contains, among others, the *SelectBooks*, *Pay* and *Ship* activities, which will need to be realized by connecting them to the *SelectBooks*, *Pay* and *Ship* concerns that are shown in the middle of the figure.

An example of the latter is present in Figure 2 as well: neither the *SelectBooks*, *Pay* nor *Ship* concerns contain any reference to the *Report* concern, whereas an unanticipated concern connection can be made between the *Report* concern and those three concerns.

Unify allows specifying both anticipated and unanticipated connections using its *connector* construct. In our example, the following connectors can, among others, be used to connect the different concerns:

1. An **activity connector** can be used to specify that the *SelectBooks* activity in the *OrderHandling* concern should be executed by executing the *SelectBooks* concern (and likewise for the *Pay* and *Ship* activities and concerns). Thus, activity connectors allow hierarchically decomposing workflows into different concerns.
2. An **after connector** can be used to specify that the *Report* concern should be executed after the *Confirm* activity in the *SelectBooks* concern, after the three *Payment* activities in the *Pay* concern, and after the two *Ship* activities in the *Ship* concern. Thus, after connectors allow expressing the *after* pattern that is currently offered by aspect-oriented approaches.
3. A **parallel connector** can be used to specify that the *SavePreference* concern should be executed in parallel with the *SelectBook* and *AddBook* activities in the *SelectBooks* concern. Thus, parallel connectors allow expressing a pattern that is not currently offered by aspect-oriented approaches.
4. A **free connector** can be used to specify that the *VerifyBankAccount* concern should be executed after the *OrderHandling* concern's *Pay* activity has been executed and before its *Ship* activity is executed. Thus, free connectors allow invasively changing a concern's control flow by introducing additional splits and joins, e.g., in order to synchronize two parallel branches.

Activity connectors express anticipated concern connections, while the other connectors express unanticipated concern connections. The Unify connector mechanism, which offers other connectors in addition to the ones mentioned above, and which is discussed in Section 5, defines the abstract syntax of our connectors.

We have defined a textual concrete syntax for our connectors, which is available in Backus–Naur form at [11]. Listing 1 shows how the above activity, after, and free connectors can be expressed using this syntax. If one would apply all the above connectors to the concerns of Figure 2, one would obtain the workflow of Figure 1.

4 The Unify Base Language

Unify is designed to be applicable to a range of concrete workflow languages, as long as they conform to a number of basic assumptions. These assumptions are expressed as a meta-model for our workflow concerns. We do not restrict ourselves to any particular concrete workflow language as long as it can be defined as an extension to this meta-model. The meta-model allows expressing *arbitrary workflows* [12], i.e., workflows whose control flow is not restricted to

```
SelectBooksConnector:
CONNECT OrderHandling.SelectBooks TO SelectBooks

ReportConnector:
CONNECT Report AFTER activity("SelectBooks\.Confirm|Pay\..*Payment|Ship\.ShipBy.*")

VerifyBankAccountConnector:
CONNECT VerifyBankAccount
AND-SPLITTING AT controlport(OrderHandling.Pay.ControlOut)
JOINING AT controlport(OrderHandling.Ship.ControlIn)
```

Listing 1. Example activity, after, and free connectors

a predefined set of control flow patterns, and is therefore also compatible with more restricted workflows such as *structured workflows*.

Figure 3 provides the meta-model for our workflow concerns. This meta-model does not contain the Unify connector mechanism, which is given in Section 5. The complete Unify meta-model is the union of these two meta-models. The meta-models are expressed using UML, with well-formedness constraints specified in OCL.

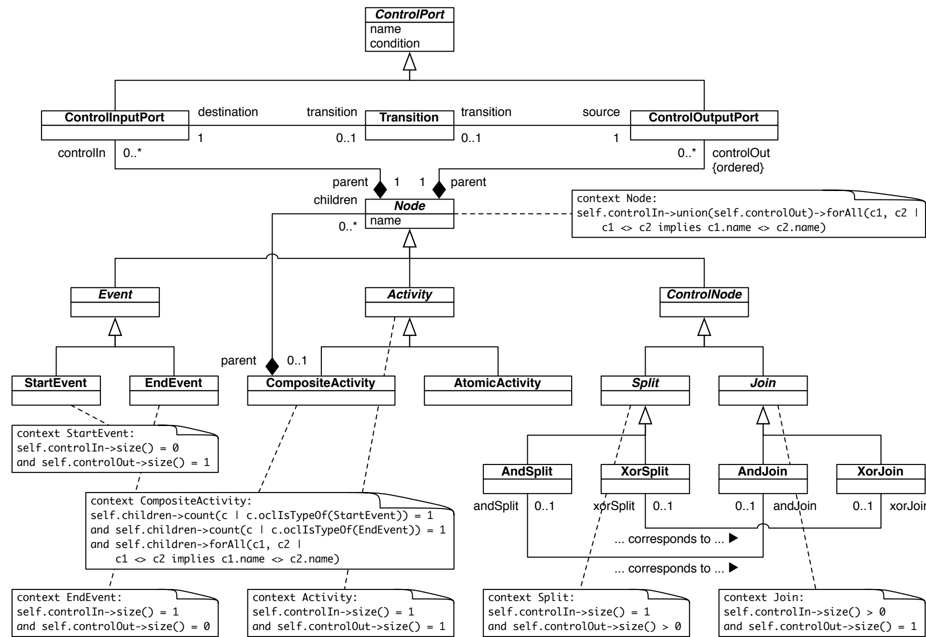

Fig. 3. The Unify base language meta-model

A workflow concern is modeled as a *CompositeActivity*. Each *CompositeActivity* has the following children: (1) A *StartEvent*, which represents the point where

the *CompositeActivity*'s execution starts. (2) An *EndEvent*, which represents the point where the *CompositeActivity*'s execution ends. (3) Any number of *Activities*, which are the units of work that are performed by the *CompositeActivity*. (4) Any number of *ControlNodes*, which are used to route the *CompositeActivity*'s control flow. (5) 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 meta-model supports the *basic control flow patterns* [13], which are sufficient for expressing most workflows. It does 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 and BPMN workflow languages.

5 The Unify Connector Mechanism

The Unify connector mechanism is based on aspect-oriented principles [2]. It allows adding the functionality defined by a certain workflow concern (which is modeled as a *CompositeActivity*) at certain locations in another workflow concern. In aspect-oriented terminology, the former concern is the *advice*, while the latter is the base concern. The locations where the advice is added are called *joinpoints*, and are either the base concern's activities, splits, or control ports. The process of adding the functionality to the base concern is called *weaving*.

Unify promotes separation of concerns by allowing workflow concerns to be specified in isolation of each other, as separate *CompositeActivities*. These can be executed separately, or can be connected to other concerns using connectors. Figure 4 shows the meta-model for the Unify connector mechanism. In the interest of brevity, the definition of 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.

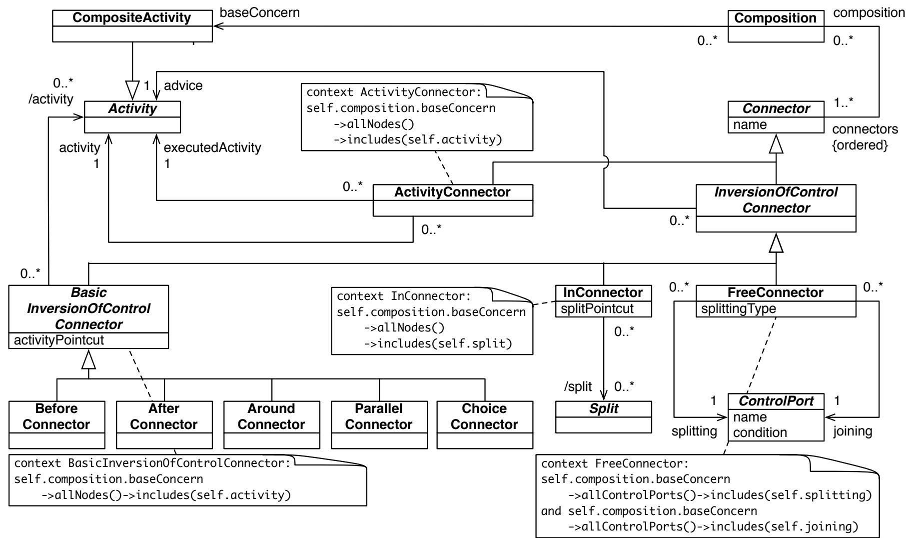

Fig. 4. The Unify connector language meta-model

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 the connectors 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 within the specification of a concern where extra functionality — the *advice* — can be inserted using an *InversionOfControl-Connector*. Joinpoints in existing aspect-oriented approaches for workflows are either every XML element of the workflow definition [6] or every workflow activity [3]. As is shown in Table 1, our approach supports three kinds of joinpoints: *Activities*, *Splits*, and *ControlPorts*. The joinpoint model is static, which has the advantage of allowing us to define a clear weaving semantics (see Section 6.3).

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. This allows specifying sets of joinpoints as identifier patterns. Pointcuts can be expressed using the predicates in Table 2.

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 [3].

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 *Choice-Connectors* 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.

In order to widen the applicability of our approach, our connector mechanism is defined in terms of our base language meta-model, which may be extended towards different concrete workflow languages. Thus, we assume that there are no prohibitive semantic differences between the way in which these languages' concepts are mapped to our meta-model. As there is a consensus on the semantics of the basic control flow patterns within the workflow community, this seems to be a safe assumption. The way in which semantic details may be added to our approach is an interesting avenue for future work, and will become more relevant when more advanced control flow patterns are added to the base language meta-model.

6 Discussion

6.1 Interaction with the Control Flow Perspective

The connector mechanism described above allows invasively changing a base concern by connecting other concerns to it. In this subsection, we focus on the effects of the connector mechanism on a base concern's control flow. Our goal here is to prevent that connecting a well-behaved concern to a well-behaved base concern results in a composition that is not well-behaved. A concern is well-behaved if it can never deadlock nor result in multiple active instances of the same activity [12].

Before-, *After-*, *Around-*, *Parallel-* or *ChoiceConnectors* cannot negatively influence a base concern's control flow because they merely result in the execution of some extra behavior around the joinpoint activity. Thus, they cannot influence any part of the base concern's control flow other than the execution of the joinpoint activity itself. *InConnectors* cannot negatively influence a base concern's control flow because they merely result in the addition of an extra branch to an existing split. Thus, they cannot influence any part of the base concern's control flow other than the execution of the split itself. Therefore, we will only discuss the effects of the *FreeConnector*.

Existing research [12] considers three kinds of workflows with respect to the structure of their control flow: *arbitrary workflows*, *structured workflows*, and

restricted loop workflows. In general, the Unify base language allows expressing arbitrary workflows. However, a specific extension of the Unify meta-model may be more restrictive. Therefore, we restrict the *FreeConnector* depending on the kind of workflows that is supported by the current extension.

Arbitrary Workflows. Intuitively, arbitrary workflows are workflows in which a split does not need to have a corresponding join. There is no guarantee that every arbitrary workflow is well-behaved; deciding whether a given arbitrary workflow is well-behaved requires the use of verification techniques [12].

Using a free connector to connect a well-behaved arbitrary workflow concern to a well-behaved arbitrary base workflow concern may give rise to a composition that is not well-behaved, but it is not possible to prevent this without introducing structure into the arbitrary workflow concerns. Therefore, we do not introduce any restrictions on the free connector when it is used to connect arbitrary workflow concerns. However, the semantics of our connector mechanism (see Section 6.3) allows applying the same verification techniques that are used to decide whether the connected workflow concerns are well-behaved to the composition of these workflow concerns, and the connector mechanism thus does not introduce any new challenges in this regard.

Structured Workflows. Intuitively, structured workflows are workflows in which every AND-split has a corresponding AND-join, and every XOR-split has a corresponding XOR-join. Thus, each split and its corresponding join constitute a block structure, and control can only enter or exit this block structure through the join or split. Control can also not cross the different branches of the block structure. These restrictions guarantee that a structured workflow is well-behaved [12].

Because free connectors insert a new split and and a new join into a base workflow concern in order to execute another workflow concern as an additional branch, this new branch may make a structured base workflow concern unstructured. In order to prevent this, the splitting and joining control ports of a free connector must be part of the same branch of the same block structure in case of structured workflows. Note that this restriction precludes the use of the free connector that was introduced in Section 2 in order to synchronize two parallel branches: as the splitting and joining joinpoints of this free connector are located in *different* branches of the same block structure, this connector is disallowed when the current extension only supports structured workflows.

Restricted Loop Workflows. Intuitively, restricted loop workflows are workflows in which only loops need to be structured (i.e., a split must only have a corresponding join if it introduces a loop in the workflow). Thus, only loops constitute block structures. Restricted loop workflows are less expressive than arbitrary workflows and more expressive than structured workflows. Depending on the implementation of the underlying workflow engine, it can be guaranteed that restricted loop workflows are well-behaved [12]. Similar to our approach for structured workflows, we restrict the free connector in case of restricted loop workflows: the splitting and joining control ports of a free connector must be part of the same branch of the same block structure.

The above restrictions aim to *prevent* undesirable effects of using the Unify connector mechanism. One can also envision approaches that *detect* undesirable effects. For example, in previous work [14] we have designed and implemented a means of expressing and statically verifying control flow policies for Unify workflows.

6.2 Interaction with the Data Perspective

In addition to effects on the control flow perspective of the connected workflow concerns, the connector mechanism has effects on the data perspective of the connected workflow concerns. For example, a connected workflow concern may reference a variable that is not defined at the place where it is woven. The effects of the connector mechanism on the data perspective depend on the approach that is used by the specific extension to the Unify meta-model to pass data from one activity to another. Existing research [15] has identified the following approaches: *integrated control and data channels*, *distinct control and data channels*, and *no data passing*. We assume that both of the connected workflow concerns use the same approach.

Integrated Control and Data Channels. In this approach, control flow and data are passed simultaneously between activities, and transitions are annotated with which data elements must be passed. Activities can only access data that has been passed to them by an incoming transition. Given two workflow concerns that use this approach, we must make sure that the weaving of the two workflow concerns results in a correct composition with regard to the data elements. Therefore, a connector must specify which data elements should be passed from the base workflow concern to the other workflow concern and back. These data elements must be accessible at the joinpoint. The weaving process will generate transitions from the base workflow concern to the other workflow concern and back, and annotate these transitions with the data elements to be passed, resulting in a correct composition.

Distinct Control and Data Channels. In this approach, data is passed between activities via explicit data links that are *distinct* from control flow links (i.e., transitions). Therefore, a connector must specify which data elements should be passed from which activities in the base workflow concern to the other workflow concern and back. The weaving process will generate transitions between the workflow concerns as usual, but will also generate distinct data links from the specified activities in the base workflow concern to the other workflow concern and back, resulting in a correct composition.

No Data Passing. In this approach, activities share the same data elements, typically via access to some common scope. Thus, no explicit data passing is required. In order to implement our connector mechanism, we could merely weave a workflow concern into the base workflow conern without any regard to the data perspective. The activities of the former workflow concern could then access all the data elements that are accessible at the joinpoint. However, this would be undesirable as it would amount to dynamic scoping: a woven workflow concern might execute correctly at one joinpoint, and not at another, depending on which

variables are accessible. Therefore, a connector must specify the data elements that are expected from the base workflow concern, and how these map to the data elements used in the other workflow concern. The weaving process can then verify whether all data is correctly mapped, and copy the data elements from the base workflow concen to the other workflow concern according to this mapping.

The solutions for each of these three approaches can be defined as extensions to the Unify meta-model. In the context of our extension towards WS-BPEL, we have already defined the extension for the *no data passing* approach. This extension encompassed associating every *CompositeActivity* with a *Scope*, which defines any number of *Variables*. This information can then be used by the weaving process.

6.3 Semantics

Because inversion-of-control connectors can invasively change the behavior of a concern by connecting another concern to it at an unanticipated location, it is important that the semantics of these connectors is clearly defined. Therefore, we have defined the semantics of the connector weaving using the *graph transformation* formalism [16]. In the interest of brevity, this section briefly discusses this semantics. We refer the reader to [17] for a complete description of our connector weaving semantics.

The semantics of a connector, which connects an workflow concern to a base workflow concern, is given by constructing a new concern that composes the base concern and the other 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 $h(R)$ (more details can be found in [16]).

In our approach, the type graph represents the meta-model shown in Figure 3. 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 well-formedness constraints can be formalized by graph constraints. Graph constraints allow the expression of properties over graphs (more details can be found in [18]).

For each possible combination of connector type (cf. Figure 4) and pointcut predicate (cf. Table 2), we specify a *composition rule*. Due to space restrictions

we only explain the rules for the *FreeConnector*. The complete set of composition rules can be found in [17].

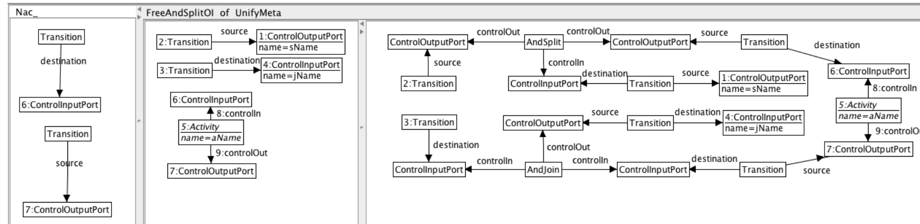

Fig. 5. The *FreeAndSplittingOI* graph transformation rule in AGG. The leftmost pane represents a NAC, which should be seen as a forbidden structure. The next pane represents the positive part of the rule's left-hand side. The rightmost pane represents the right-hand side of the rule.

Figure 5 shows the rule that corresponds to an AND-splitting *FreeConnector* expressed in the general-purpose graph transformation tool *AGG*.[2] The left-hand side of a graph transformation rule is composed of a positive condition, i.e., the presence of certain combinations of nodes and edges, and optionally, a set of negative application conditions (NACs), i.e., absence of certain combinations of nodes and edges. On the right-hand side of the transformation rule the result of weaving the workflow concern in the base workflow concern is shown. Remark that eight rules are specified for the *FreeConnector*: four rules for the AND-splitting *FreeConnector*, and four for the XOR-splitting *FreeConnector*. Each of these has four rules because of the possible combinations of control input and output ports.

The *FreeAndSplittingOI(sName : String, jName : String, aName : String)* rule adds a split at a certain control output port, adds a join at a certain control input port, and inserts an activity between the new split and join. 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 *Activity* that is to be inserted with its *ControlInputPort* and *ControlOutputPort*. The right-hand side specifies the graph after inserting the activity. 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* that is connected to the *ControlInputPort* of the inserted *Activity*. The *ControlOutputPort* of the inserted *Activity* is connected to a new *AndJoin* through a new *Transition*. The other incoming *Transition* of the *AndJoin* is the

[2] See `http://tfs.cs.tu-berlin.de/agg/`.

joining *ControlInputPort*'s original incoming *Transition*. Finally, the *AndJoin* is connected to the joining *ControlInputPort* through a new *Transition*.

7 Implementation

We have created a proof-of-concept implementation for Unify, which is available for download at [11]. The architecture of this implementation is shown in Figure 6. At the heart of the architecture lies a Java implementation of the Unify base language (cf. Figure 3) and connector mechanism (cf. Figure 4). The Unify API allows constructing and manipulating workflow concerns in-memory, while extensions to the Unify framework provide parsers and serializers for existing concrete workflow languages. A composition specifies which concerns should be loaded and which connectors should be applied to them.

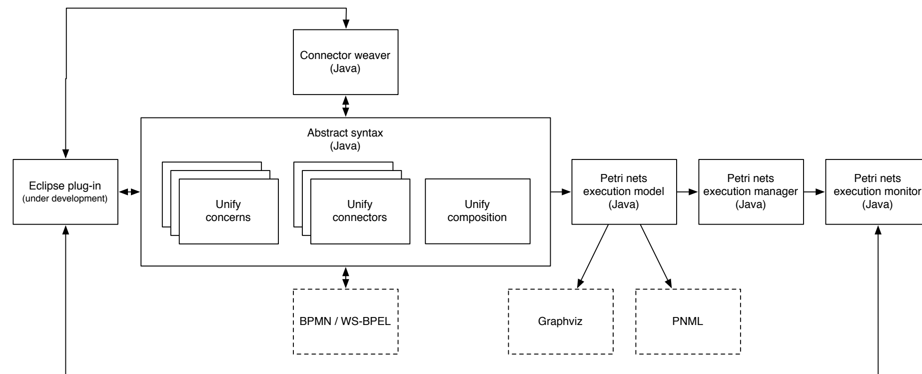

Fig. 6. Architecture of the Unify implementation

One by one, the Unify connector weaver applies the connectors to the base concern in the order specified by the composition. For each connector, the base concern is modified accordingly. The final modified base concern is transformed into a Petri nets execution model if one wants to use Unify's built-in workflow engine, or is exported back to the workflow language in which the original concerns were specified. In this latter case, the composition is serialized into a single workflow in which all concerns are woven together. An Eclipse plug-in that facilitates interaction with the Unify tool chain is currently under development.

The instantiation process is straightforward for common workflow concepts, i.e., activities and basic control flow concepts. However, three limitations may arise when instantiating Unify: (1) The base language is graph-based, which means that block-structured constructs such as those encountered in WS-BPEL should be correctly mapped to Unify's graph-based constructs, which is feasible. (2) The base language only provides the basic control flow patterns identified in existing research, which means that it is cumbersome to implement advanced

control flow patterns such as those encountered in YAWL [19]. (3) The base language focuses on the control flow perspective, which means that it provides no support for other perspectives, such as the exception handling perspective. These limitations are the result of the deliberate choice to focus on the expressiveness of the modularization mechanism rather than on the expressiveness of the individual modules, and could be addressed by iterating over the base language meta-model. We believe that the current meta-model is sufficient for demonstrating our contributions to the modularization of workflow concerns.

8 Related Work

Broadly speaking, the related work we consider can be divided into four domains:

Component Based Systems. Component based software development (CBSD) aims to promote separation of (non-crosscutting) concerns by allowing the composition of independent software components. Although some component frameworks allow modularizing specific crosscutting concerns using constructs like deployment descriptors, a general modularization mechanism for crosscutting concerns is typically unavailable. In the context of CBSD, connectors are often used to specify the roles that different software components fulfill in a composition [20]. Unify connectors are inspired by such component-based connectors, and are similarly used to specify, at deployment time, how different concerns should be composed. However, unlike component-based connectors, with respect to the connected concerns, Unify connectors describe both anticipated and non-anticipated (e.g., aspect-oriented) connections.

Traditional Workflow Languages. The most well-known current workflow languages are WS-BPEL [10] and YAWL [19]. WS-BPEL has notoriously poor support for separation of concerns (which has led to a number of aspect-oriented approaches that aim to remedy this; see below): a WS-BPEL process is a monolithic XML file that cannot be straightforwardly divided into subprocesses. YAWL improves on this in that it allows workflows to be divided into reusable sub-workflows.

Aspect-Oriented Programming for Workflows. The lack of modularization mechanisms in traditional workflow languages, most notably WS-BPEL, has led to the development of a number of aspect-oriented approaches for worfklows. AO4BPEL [6], the approach by Courbis & Finkelstein [7], and Padus [3] are the most well-known. They all allow modularizing crosscutting concerns in WS-BPEL workflows using separate aspects. Unify improves on them by allowing the modularization of *all* workflow concerns, i.e., not only crosscutting ones, and by introducing workflow-specific advice types in addition to the classic before, after, and around advice types. Moreover, Unify is not restricted to any concrete workflow language such as WS-BPEL.

Dynamic Workflow Systems. Existing research has produced a taxonomy of workflow flexibility [21]. With regard to this taxonomy, Unify mainly aims to improve workflow flexibility *by design* by providing a more expressive modularization mechanism than those offered by current workflow languages.

The use of the standard Unify connector weaver precludes other forms of flexibility, i.e., flexibility by deviation, by underspecification, or by change (which describe different kinds of runtime adaptation of workflows). Extending our execution model with support for runtime enabling and disabling of connectors would remove this restriction, but is currently beyond the scope of our research, as runtime adaptation does not improve the design or reuse of workflow concerns. This constitutes an important difference in focus with regard to workflow systems that allow dynamically changing workflows.

9 Conclusions and Future Work

Existing workflow languages have insufficient support for separation of concerns. This can make workflows hard to comprehend, maintain, and reuse. We address this problem by introducing Unify, a framework that allows specifying both regular and crosscutting workflow concerns in isolation of each other. The Unify connector mechanism allows connecting these independently specified concerns using a number of workflow-specific connectors.

Activity connectors allow expressing that an existing activity in one concern should be implemented by executing another concern, in a way that minimizes dependencies between these concerns and thus facilitates their independent evolution and reuse. Additionally, inversion-of-control connectors allow augmenting a concern with other concerns that were not considered when it was designed, and again facilitates independent evolution and reuse of these concerns.

At the heart of Unify lies a meta-model that allows expressing arbitrary workflows, and which can be mapped to several concrete workflow languages and notations. We also provide a meta-model for the connector mechanism, and discuss its interaction with the control flow and data perspectives. We provide a semantics for the weaving of the connectors using the *graph transformation* formalism.

We have identified the following directions of future work: (1) Unify connectors are currently specified using a textual syntax (cf. Section 3). We are investigating how we can support workflow developers in specifying connectors in a visual way. (2) Although a proof-of-concept implementation of Unify has been developed, tool support should be extended in order to facilitate adoption of the approach. Therefore, we are developing an Eclipse plug-in that facilitates interaction with the Unify tool chain. (3) We are working on a more extensive validation of our approach, based on the refactoring of a real-life workflow application using Unify.

References

1. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM **15**(12) (1972) 1053–1058
2. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. Lecture Notes in Computer Science **1241** (1997) 220–242

3. Braem, M., Verlaenen, K., Joncheere, N., Vanderperren, W., Van Der Straeten, R., Truyen, E., Joosen, W., Jonckers, V.: Isolating process-level concerns using Padus. Lecture Notes in Computer Science **4102** (2006) 113–128
4. Joncheere, N., Deridder, D., Van Der Straeten, R., Jonckers, V.: A framework for advanced modularization and data flow in workflow systems. Lecture Notes in Computer Science **5364** (2008) 592–598
5. Arsanjani, A., Hailpern, B., Martin, J., Tarr, P.: Web services: Promises and compromises. ACM Queue **1**(1) (2003) 48–58
6. Charfi, A., Mezini, M.: Aspect-oriented web service composition with AO4BPEL. Lecture Notes in Computer Science **3250** (2004) 168–182
7. Courbis, C., Finkelstein, A.: Towards aspect weaving applications. In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005). St. Louis, MO, USA, ACM Press (2005) 69–77
8. Gonzalez, O., Casallas, R., Deridder, D.: MMC-BPM: A domain-specific language for business process analysis. Lecture Notes in Business Information Processing **21** (2009) 157–168
9. Object Management Group: Business Process Model and Notation, version 2.0 (2011) `http://www.omg.org/spec/BPMN/2.0/`.
10. Jordan, D., Evdemon, J., et al.: Web Services Business Process Execution Language, version 2.0 (2007) `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.
11. Joncheere, N., et al.: The Unify framework (2009) `http://soft.vub.ac.be/~njonchee/artifacts/unify/`.
12. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.: On structured workflow modelling. Lecture Notes in Computer Science **1789** (2000) 431–445
13. Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N.: Workflow control-flow patterns: A revised view. BPM Center Report BPM-06-22, BPM Center (2006)
14. De Fraine, B., Joncheere, N., Noguera, C.: Detection and resolution of aspect interactions in workflows. Technical Report SOFT-TR-2011.06.20, Vrije Universiteit Brussel, Software Languages Lab, Brussels, Belgium (2011) `http://soft.vub.ac.be/~njonchee/publications/TR20110620.pdf`.
15. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow data patterns. QUT Technical Report FIT-TR-2004-01, Queensland University of Technology (2004)
16. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations. World Scientific, River Edge, NJ, USA (1997)
17. Joncheere, N., Van Der Straeten, R.: Semantics of the Unify composition mechanism. Technical Report SOFT-TR-2011.04.15, Vrije Universiteit Brussel, Software Languages Lab, Brussels, Belgium (2011) `http://soft.vub.ac.be/~njonchee/publications/TR20110415.pdf`.
18. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Monographs in Theoretical Computer Science. Springer (2006)
19. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. Information Systems **30**(4) (2005) 245–275
20. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River, NJ, USA (1996)
21. Schonenberg, H., Mans, R., Russell, N., Mulyar, N., van der Aalst, W.M.P.: Process flexibility: A survey of contemporary approaches. Lecture Notes in Business Information Processing **10** (2008) 16–30