Vrije Universiteit Brussel

Detection and Resolution of Aspect Interactions in Workflows

Technical Report SOFT-TR-2011.06.20

Bruno De Fraine, Niels Joncheere, and Carlos Noguera
Vrije Universiteit Brussel
Software Languages Lab
Pleinlaan 2, 1050 Brussels, Belgium
{bdefrain,njonchee,cnoguera}@vub.ac.be

Detection and Resolution of Aspect Interactions in Workflows

Bruno De Fraine, Niels Joncheere, and Carlos Noguera
Vrije Universiteit Brussel
Software Languages Lab
Pleinlaan 2, 1050 Brussels, Belgium
`{bdefrain,njonchee,cnoguera}@vub.ac.be`

June 20, 2011

Abstract

Workflow systems have become a popular means of automating processes in many domains. Current workflow languages, however, provide only limited modularization mechanisms, and thus suffer from a lack of separation of concerns. Inspired by aspect-oriented research, several extensions to workflow languages have been developed which allow modularizing workflow concerns into separate aspects. Unfortunately, these extensions suffer from similar feature interaction problems as general-purpose aspect-oriented programming languages. In previous work, we have outlined a strategy for managing such interactions. This technical report extends and applies these ideas to workflows. Control flow policies are specified using high-level predicates, and are verified by translating both these predicates and the workflows on which they are specified to DFAs. By computing the intersection between a policy DFA and a workflow DFA, we can determine whether a given policy holds for a given workflow.

Contents

1 History

Date	**Comment**
June 20, 2011	Creation

2 Introduction

Workflow languages are increasingly being used to describe the composition of certain components or (web) services, and how they interact. This is accomplished by dividing business processes into activities, and specifying the ordering in which these activities are to be executed. This ordering is called the control flow perspective, as it describes how control flows between the activities. Typically, control can be split into several branches and joined at a later time. This allows specifying parallelism and choice.

Historically, workflow languages offered limited support for dividing workflows into separate modules. This hampered separation of concerns. Based on ideas that originated in the aspect-oriented programming community [12], several aspect-oriented extensions were developed which allow modularizing crosscutting workflow concerns into separate aspects [3, 4, 2].

However, these aspect-oriented approaches for workflows suffer from similar feature interaction problems as traditional aspect-oriented programming languages. In previous work [5], we have outlined a strategy for managing such interactions. This was accomplished by producing control flow graphs and call graphs of an application with woven aspects, and allowing the verification of control flow policies using these graphs.

The goal of this technical report is to extend and apply the ideas of [5] to workflows. This seems natural, as the control flow perspective is more explicit in workflow languages than in general-purpose languages. However, the control flow is also more complex, in that it natively supports parallelism. On the one hand, we allow specifying control flow policies using high-level predicates that are similar to those proposed in [5]. These predicates are translated into regular expressions, which are in turn translated into deterministic finite automata (DFAs). On the other hand, we translate (woven) workflows into DFAs as well. By computing the intersection between a policy DFA and a workflow DFA, we can then determine whether the given policy holds for the given workflow.

The outline for this technical report is as follows. Section 3 introduces UNIFY, a workflow system whose meta-model we use to represent workflows. Section 4 describes our policy language. Section 5 describes how our policies can be verified using DFAs, and Section 6 states our conclusions and future work.

3 Aspects for Workflows using UNIFY

3.1 Context

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 specifying the ordering in which these activities need to be performed. This ordering is called the control flow perspective, as it describes how control flows between the activities. Typically, control can be split into several branches and joined at a later time. This allows specifying parallelism and choice.

Realistic workflows consist of several *concerns*, which are connected in order to achieve the desired behavior. However, if all of these concerns can only be specified in a single, monolithic workflow specification, it will be hard to add, 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 dubbed *crosscutting concerns* [12], and this problem has given rise to *aspect-oriented programming* for workflows [3, 4, 2].

3.2 Motivation

Consider the workflow in Figure 1, which is a simplified version of an automated process that is performed by the ERP system of one of our industrial partners. The workflow specifies how the company's IT department performs an IT project, and is expressed using BPMN [15], which is a standard for visual modeling of workflows. The workflow starts at the start event, and performs a feasibility study. Based on this study, the decision is made to go ahead with the project or not. If the project is rejected, the customer is provided with the reason why, and the workflow reaches the end event. If the project is approved, a traditional waterfall approach to software development is taken. A reporting activity is situated after each of the phases, and feedback loops allow returning to a previous phase. Billing is performed right before the end of the workflow.

Like any realistic software application, this workflow consists of a number of *concerns* that are connected in order to achieve the workflow's desired behavior. We can easily identify the "software development", "reporting", and "billing" concerns which occur at various places across the workflow. The general software engineering principle of *separation of concerns* [13] 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 WS-BPEL [1] workflow is expressed in a single, monolithic XML file that cannot be straightforwardly divided into sub-workflows. YAWL [14] supports sub-workflows, but workflows are flattened into a single XML file before deployment. 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

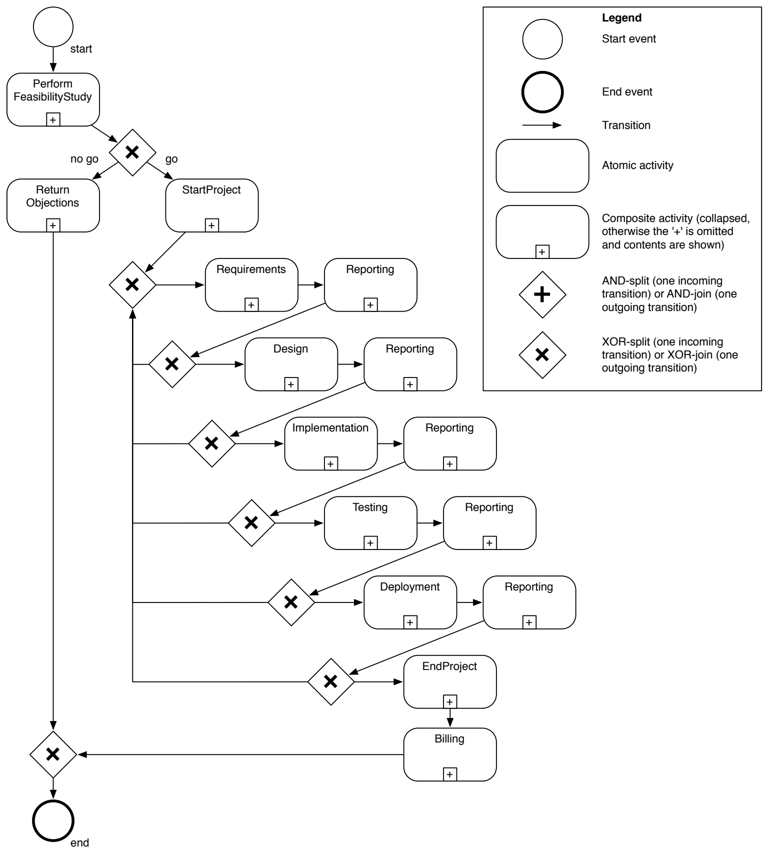


Figure 1: Example workflow

improved. In the workflow in Figure 1, one could for example vary the behavior of the workflow by deploying a different “design” sub-workflow in different situations.

A second kind of connection between concerns is useful when concerns *crosscut* a workflow: some concerns cannot be cleanly modularized 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 [12]. Existing aspect-oriented extensions to WS-BPEL, such as AO4BPEL [3], allow specifying crosscutting concerns in separate aspects. These can then be connected to the main workflow *before*, *after*, or *around* a certain set of activities. 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. If all concerns, crosscutting or otherwise, are specified using the same construct, reusability is improved. We call such an approach a *uniform* approach.

Moreover, the aspect-oriented extensions mentioned above only support the basic connection patterns (*before*, *after*, and *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 control flow patterns such as parallelism, choice, and loops. For example, the before, after, or around patterns do not allow inserting a feedback loop from the `ReturnObjections` activity in Figure 1 to the `PerformFeasibilityStudy` activity, which would allow renegotiating the IT project. Workflow languages would benefit from support for such workflow-specific concern connection patterns.

Furthermore, these aspect-oriented extensions are all aimed at WS-BPEL, and cannot easily be applied 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. More variability in terms of the applicable languages and possible implementation techniques would make a modularization approach more widely applicable.

In order to maximize the expressiveness of the connector mechanism, the control flow patterns should not be constrained to a predetermined set of connections. The patterns should provide a high degree of freedom to enable a developer to formulate solutions to unanticipated situations. Of course this flexibility and freedom may never cause undesirable behavior such as deadlocks and unreachable paths. The workflow developer can be supported in creating safe workflows by building on a long tradition of analysis and verification techniques for workflows.

3.3 UNIFY

In order to tackle the above problems, we have proposed UNIFY [10, 11], a framework for modularization of workflow concerns. In this technical report, we employ UNIFY mainly because of its base language meta-model, which is representative of current workflow languages, and is shown in Figure 2.

The main contributions of UNIFY are the following:

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 [8]. UNIFY, on the other hand, provides 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 meta-model.

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

5. UNIFY provides a meta-model and an implementation which enables the introduction of new patterns and connectors as they arise. Moreover, UNIFY can be implemented using multiple strategies. For example, it does not require a modified workflow engine.

UNIFY accomplishes this by allowing each workflow concern to be implemented as a separate module. Figure 2 shows the UNIFY base language meta-model. A concern is modeled using the *composite activity* construct. A composite activity consists of a *start event* and an *end event*, which represent the point where the concern starts and ends its execution, respectively. Between these two points, *nodes* can be executed, which are either *activities* or *control nodes*. Activities are a concern's units of work, and are either composite activities or *atomic activities*. Control nodes are used to split or join a concern's control flow, and can express both parallelism (AND) and choice (XOR). Nodes have *control input ports* and *control output ports*, which represent the points where control enters and exits a node, respectively. The way control flows from a concern's start event, through its nodes to an end event, is specified by connecting all nodes' control output ports to other nodes' control input ports using *transitions*. Thus, implementing a composite activity using UNIFY is similar to implementing a workflow using other workflow languages, the main difference being that using these other workflow languages, all concerns are implemented in the same workflow.

After the required concerns have been identified and implemented (or retrieved from a library of previously implemented concerns), the connections between the concerns should be specified. Based on our experiences in workflow development, we have identified five patterns according to which workflow concerns can connect, and these can be divided into two categories:

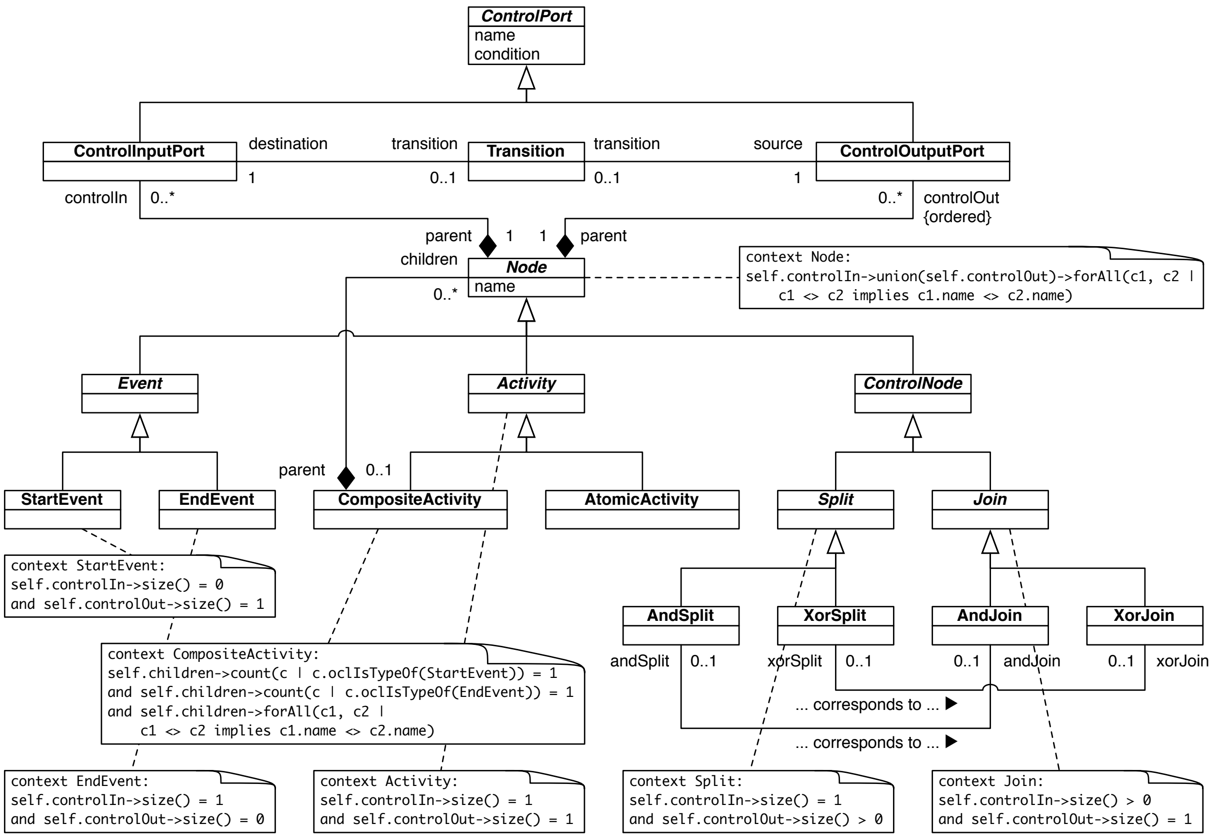
1. **Anticipated concern connections** are concern connections that are explicitly anticipated by one of the concerns: this concern is aware, at design time, of the fact that it will connect to another concern at a certain point in its execution.

   (a) The *sub-concern* pattern: one concern explicitly specifies that at a certain point in its execution, another concern should be executed. Figure 3 (first row) shows how this pattern can be used to specify that the `DesignPhase` activity in the `SoftwareDevelopment` main concern should be executed by executing the `Design` concern.

2. **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.

   (a) The *sequence* pattern: one concern should be performed in sequence with a part of another concern. Figure 3 (second row) shows how this pattern can be used to specify that the `Reporting` concern should be executed after the `DesignPhase` activity in the `SoftwareDevelopment` main concern.

   (b) The *parallelism* pattern: one concern should be performed in parallel with a part of another concern. Figure 3 (third row) shows how this pattern can be used to specify that the



```
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 2: The UNIFY base language meta-model

`TestDesign` concern (which handles the design of the integration tests that may be performed at deployment time) should be executed in parallel with the `DesignPhase` activity in the `SoftwareDevelopment` main concern. The AND-split and -join are not present in the original concern, and are added by UNIFY.

(c) The *choice* pattern: one concern should be performed as an alternative to a part of another concern. Figure 3 (fourth row) shows how this pattern can be used to specify that the `LimitedTesting` concern should be executed as an alternative to the `TestingPhase` activity in the `SoftwareDevelopment` main concern if the project is late. The XOR-split and -join are not present in the original concern, and are added by UNIFY.

(d) The *loop* pattern: one concern should introduce a loop into another concern. Figure 3 (fifth row) shows how this pattern can be used to introduce a loop into the `Renegotiation` concern. The XOR-join and -split are not present in the original concern, and are added by UNIFY.

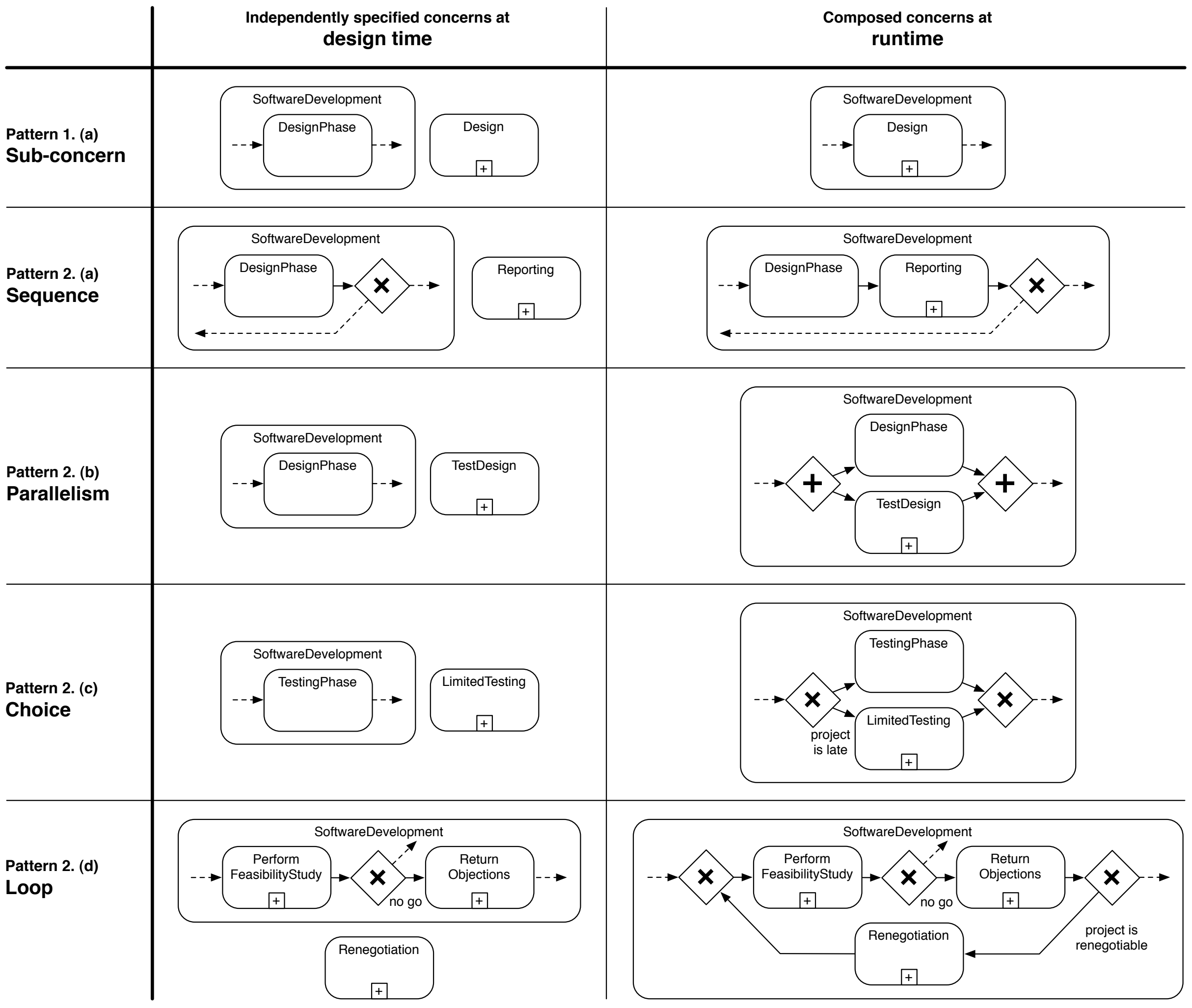
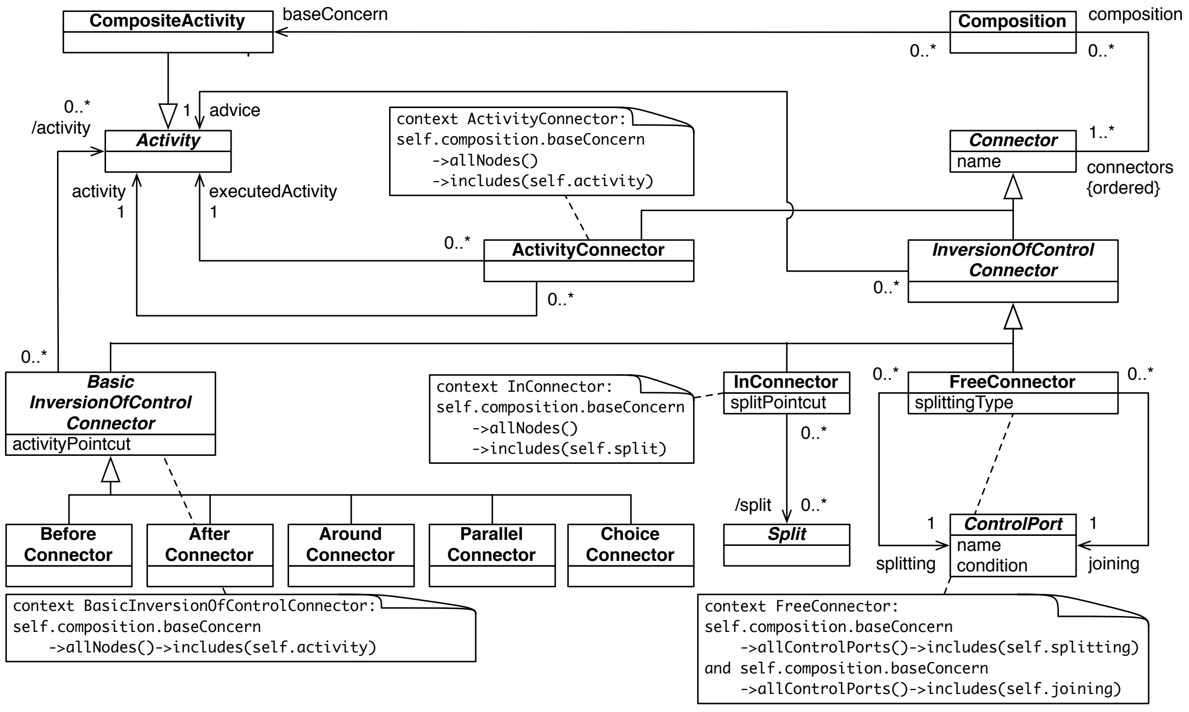


Figure 3: Examples of the five concern connection patterns

The UNIFY *connector* construct allows expressing each of these connection patterns. Figure 4 shows the UNIFY connector language meta-model, which complements the base language meta-model from Figure 2. The specific kinds of connectors offered by UNIFY are beyond the scope of this technical report, so we will not discuss them further. The interested reader is referred to [11] for an in-depth description of the UNIFY connector mechanism. A concrete workflow is created using the *composition* construct, which references the workflow's main concern, other concerns, and connectors. The other concerns can then be woven into the main concern according to the connectors using a static weaver. This results in a workflow specification that conforms to the base language meta-model from Figure 2.



```
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 4: The UNIFY connector language meta-model

4 Workflow Policies to Detect Aspect Interactions

4.1 The Problem of Aspect Interactions

The separation of concerns achieved in aspect-oriented systems implies that each aspect is described in isolation, without relation to the other aspects in the system. However, when multiple aspects are combined in the same application, one aspect may violate the (explicit or implicit) assumptions of other aspects in the system. As a result, the combined system may exhibit unintended or undesirable behavior. For example, when several aspects advise the same program point, their weaving order determines the semantics of the resulting composition and a specific ordering may be critical to obtain a correct operation [6]. The general phenomenon of aspects influencing each other's behavior is referred to as *aspect interaction* in the field of aspect orientation.

It is easy to see that the problem of aspect interactions also manifests itself in the case of aspect-oriented workflow approaches and UNIFY. As a simple example in the context of the workflow from Figure 1 on page 6, consider the unanticipated application of two concerns to the `DesignPhase` activity according to two interaction patterns from Figure 3 on page 10:

1. The `Reporting` concern should be executed immediately after the `DesignPhase` activity (according to the *sequence* pattern).
2. The `TestDesign` concern should be executed in parallel with the `DesignPhase` activity (according to the *parallelism* pattern).

At runtime, the weaving of these two additional concerns (in the specified order) results in the workflow fragment that is displayed in Figure 5. In this expanded view, a subtle interaction may be observed: the `Reporting` concern is intended to be executed immediately after the completion of the `DesignPhase` activity, however, the workflow fragment indicates that an AND-join with the `TestDesign` concern should occur first. If the `TestDesign` activity finishes before the `DesignPhase` activity, this join may occur nearly immediately. However, if the `TestDesign` activity takes longer, the reporting of the completion of the design will be unexpectedly delayed. (The difference between these two execution scenarios illustrates that some aspect interaction problems only manifest themselves during specific executions and not in every execution of the composed workflow.) We argue that it is highly desirable to support the workflow developer in the management of such aspect interactions. The remainder of this technical report will propose an approach that provides this support.

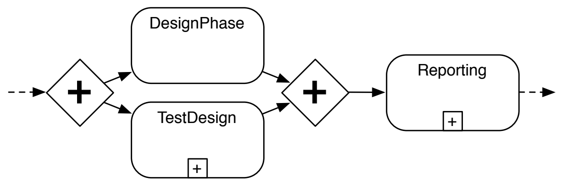


Figure 5: Application of `Reporting` and `TestDesign` concerns to the `DesignPhase` activity

4.2 Outline of the Approach

In previous work [5], we have proposed a general technique to manage aspect interactions over the course of the development of the software. This technique relies on the following elements: (i) the documentation of aspects with formal policies that encode requirements for the correct behavior of an aspect, and (ii) the automatic checking of the composed application to detect any violations of these policies. When an aspect interaction problem corresponds to the breaking of a documented assumption of one aspect by another aspect, then our technique will be able to detect this statically (*i.e.*, before the composition is executed). At this point, the developer may change the implementation of the system to remedy the problem. Of course, the successful detection of interaction problems requires that all the 'important' assumptions of an aspect are encoded as policies. While no specific guidance is offered to the developer on how specific or detailed these policies need to be, we do envision that policies may be gradually refined when an iterative software development process is used. This is outlined in Figure 6.

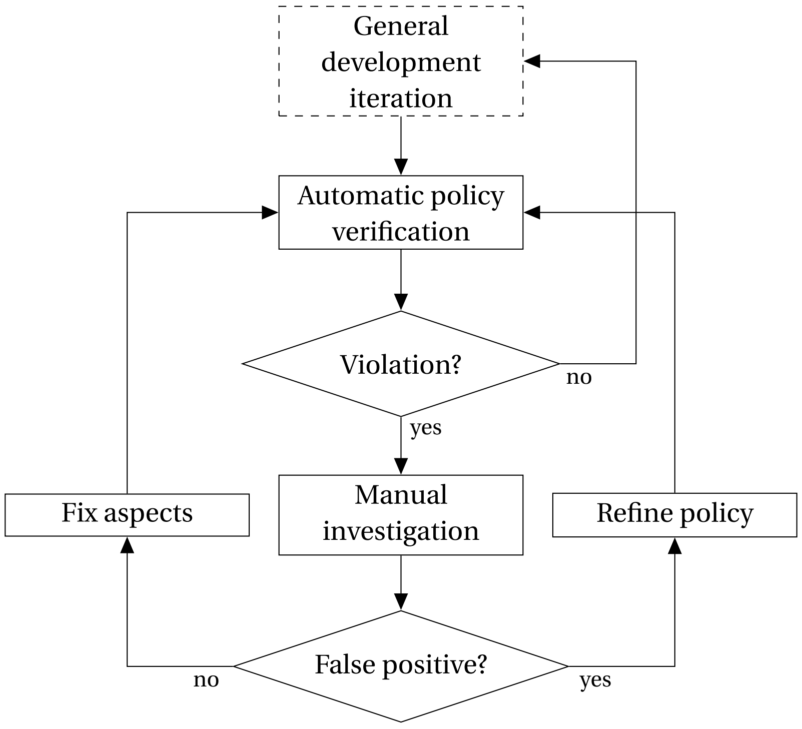


Figure 6: Flow diagram of the refinement process

Initially, the aspects may be specified with rough policies that are by default rather restrictive and that do not take other aspects into account. After each development iteration, the composition is automatically checked for violations of the policies of the involved aspects. If no policy violations are detected, we can proceed to the next iteration. Otherwise, the developer can manually investigate each of the reported violations. If a violation turns out to be an actual problem, the approach succeeded in preventing an aspect interaction problem, and the developer can fix the involved aspects or main concern implementation. If a violation is a false positive, the developer can refine the violated policy with more specific knowledge to recognize this case as a valid combination. As such, the policies will evolve in a controlled manner to more refined and permissive specifications that may contain information about other aspects in the system.

Although this approach was originally proposed in the context of general programming languages, we observe that it applies equally well for the case of workflow composition with aspects, as realized by technologies such as UNIFY. In order to instantiate the general approach for this particular context, it means we must develop the following two elements:

1. a *policy language* to define policies that have to be honored in a woven workflow

2. a method for the *automatic verification* of these policies

We will describe the policy language in the following subsection; the automatic verification of the policies is the subject of Section 5.

4.3 A Language for Workflow Policies

Workflow policies are composed by combinations of atomic predicates that express relations between activities in possible traces of execution of the workflow. Each aspect is expected to declare its assumptions about the behavior of the workflow using these predicates. We propose a workflow policy language based on predicate logic, using the five predicates detailed in Table 1. Predicates operate on (possibly singleton) sets of activities, and can be composed using classic logic connectors ($\neg$, $\wedge$, $\vee$) to specify a complete policy. Each of the predicates is explained below and an example policy for the Software Development Process workflow introduced in Section 3 is used to illustrate its utility.

Predicate	**Informal Definition**
$follows(A,B)$	An occurrence of A should be immediately followed by B
$between(A,B,C)$	A sequential occurrence of first A then B should have at least one occurrence of C in between
$may(A,B)$	There must exist an execution with a sequential occurrence of first A then B
$must(A,B)$	An occurrence of A should be sequentially followed by B
$guards(A,B)$	An occurrence of A should be sequentially preceded by B

Table 1: Predicates for workflow policies

follows(A,B) It is sometimes the case in workflows that it is necessary to assure that an activity is executed immediately after another one. Given the common and explicit use of parallelism in workflows, through the AndSplit and AndJoin control nodes in UNIFY, checking this kind of constraints may not be trivial. This is further complicated in the presence of runtime aspects that can interleave additional activities. To express this kind of constraints, we propose the $follows(A,B)$ predicate. This predicate is true when every occurrence of A is immediately followed by B. In the case of the Software Development Process workflow, we can imagine that for quality control, every development activity should be immediately followed by a Reporting activity. This constraints is expressed by the following policy:

$$follows(\texttt{Design Phase}, \texttt{Reporting})$$

between(A,B,C) A predicate to express that an activity must occur between two other activities is useful to assure that connection patterns that introduce forks (Joins and Splits) preserve certain properties. To this end, we propose the $between(A,B,C)$ that specifies that between the execution of activities A and B, C should always occur. In our example workflow, a common quality property is that implementation should not start until the design phase is finished. This can be expressed as:

$$between(\texttt{Requirements}, \texttt{Implementation}, \texttt{Design})$$

may(A,B) So far, the predicates we have introduced reason about all the possible activity execution traces of the workflow. It is some times interesting for the workflow developer to identify properties that must hold on at least a single trace. Thus, we propose the $may(A,B)$ predicate. This predicate checks that A is eventually followed by B in at least one trace. In the Software Development Process workflow of the example, it is desirable that after the objections to the feasibility study are returned there is a path that leads to renegotiation. To codify this constraint, the policy can be written as:

$$may(\texttt{Return Objections}, \texttt{Renegotiation})$$

must(A, B) Related to the may predicate, *must*(A, B) states that every A must be eventually followed B. In our example, the restriction that Reporting must immediately follow the Design Phase can be relaxed to:

$$\textit{must}(\texttt{Design Phase}, \texttt{Reporting})$$

guards(A, B) Finally, *guards*(A, B) expresses that for B to occur, A must have been executed previously; working as the converse of $must$. In the example, it can be desirable to assure that the customer is only billed when he is actually delivered artifacts. Delivery in the Software Development Process workflow is represented by the Deployment and Return Objections activities. Thus, this policy is expressed as:

$$\textit{guards}(\{\texttt{Deployment}, \texttt{Return Objections}\}, \texttt{Billing})$$

5 Automatic Verification of Policies using Finite State Automata

Workflow policies represent business rules that must be respected by a workflow. Workflow policies express expected properties of the sequences of possible executions of activities in a workflow. We choose to implement the predicates that compose workflow policies as regular expressions over the possible traces of activities defined by the workflow. Regular expressions are familiar constructions to developers, since they are present in a number of programming languages; this familiarity gives regular expressions an advantage over more powerful constructs commonly used in verification such as Linear Temporal Logic (LTL). Additionally, the algorithms for working in the realm of regular languages are well-known [9]. This includes regular expressions, but also finite state automata.

Our approach to the automatic verification of policies is outlined in Figure 7. In brief, this is a three-step process. First, the workflow is converted to a deterministic finite automaton (DFA). This occurs through a translation to a non-deterministic finite automaton (NFA) with ε-transitions, after which standard algorithms can be employed for the remainder of the conversion. Second, the policies are translated to regular expressions, which are consequently also converted to DFAs using standard algorithms. Finally, the DFAs of the workflow and the policies are intersected with each other, and the result is analyzed for satisfiable paths. Each of these steps is detailed in the following subsections.

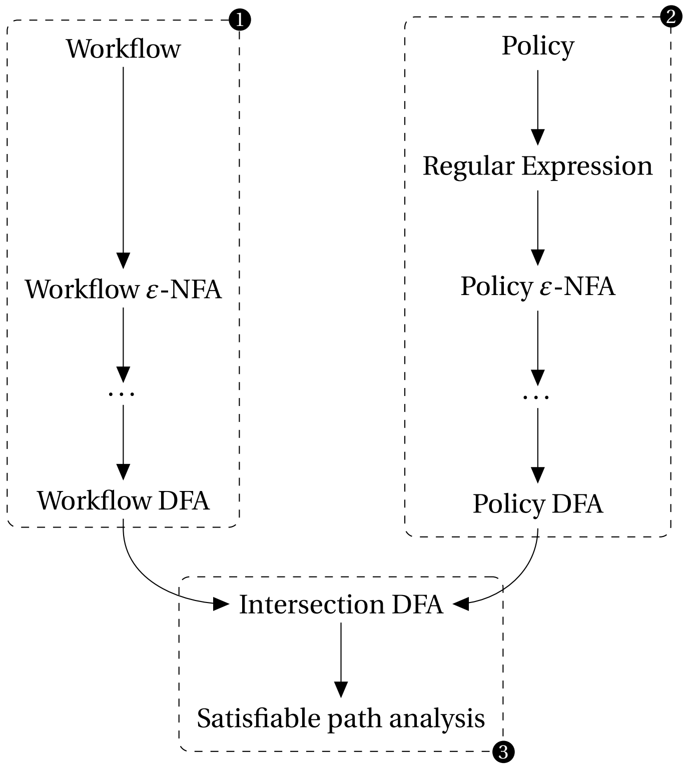


Figure 7: Overview of the verification process

5.1 Workflow to DFA

In order to simulate the behavior of a UNIFY workflow, we convert it to a deterministic finite automaton. Activities of the workflow are translated into transitions of the DFA, with states being added to the DFA so that the transitions respect the ordering of the activities in the original workflow.

5.1.1 Automaton Construction

Workflows are translated into deterministic finite automata incrementally in three steps. First, the transitions of the workflow are used to derive the states of the automaton, then transitions of the automaton are generated in function of the nodes of the workflow. This produces a non-deterministic automaton with ε-transitions. The NFA is then transformed into a DFA using well-known algorithms. Workflow patterns and their corresponding automaton translations are depicted in Figures 8 and 9.

(UNIFY) Transitions Each transition in the workflow is translated into a state of the automaton. The source and destination activities linked by the workflow transition are stored in maps.

Sequence Each sequential activity is transformed into a transition labeled with the name of the activity. The states that the automaton transition links are obtained from the maps generated from the UNIFY transitions.

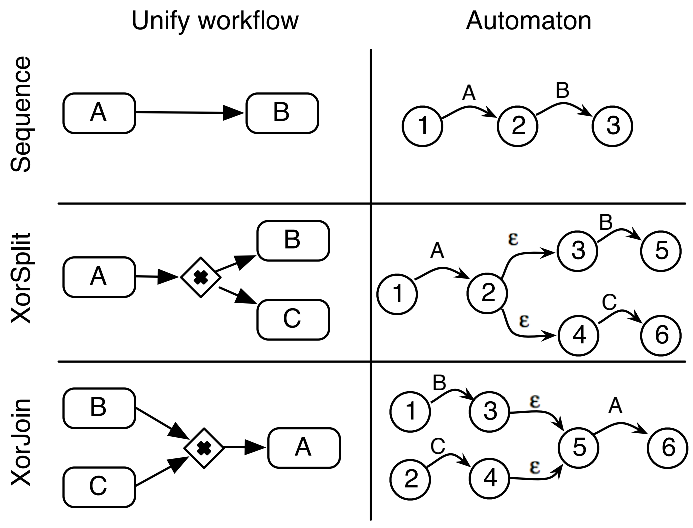


Figure 8: UNIFY workflow to automaton conversion for sequence, XorSplit, and XorJoin

XorSplits When an XorSplit is encountered, one state for each of the branches of the split is added to the automaton. The source activities map generated from the workflow's transitions is used to find the state corresponding to the activity that precedes the split. Then, ε-transitions are added from this state to each of the states of the branches of the split. Finally, each of the new states is added to the source and destination maps.

XorJoins XorJoins are handled in a similar way than that of XorSplits; a state is added to the automaton for each incoming branch, as well as ε-transitions from these states to a new collecting state. All the new states are added to the source and destination maps.

Parallelism To simulate the parallelism expressed by the use of AndSplits and AndJoins, a mechanism similar to that of XorSplit and XorJoins is used. First, ε-transitions and states are added at the split and corresponding join. Then the automata for each of the branches is generated. Then, the automata for each of the branches are interleaved. Finally, the interleaved states are connected to the rest of the automaton.

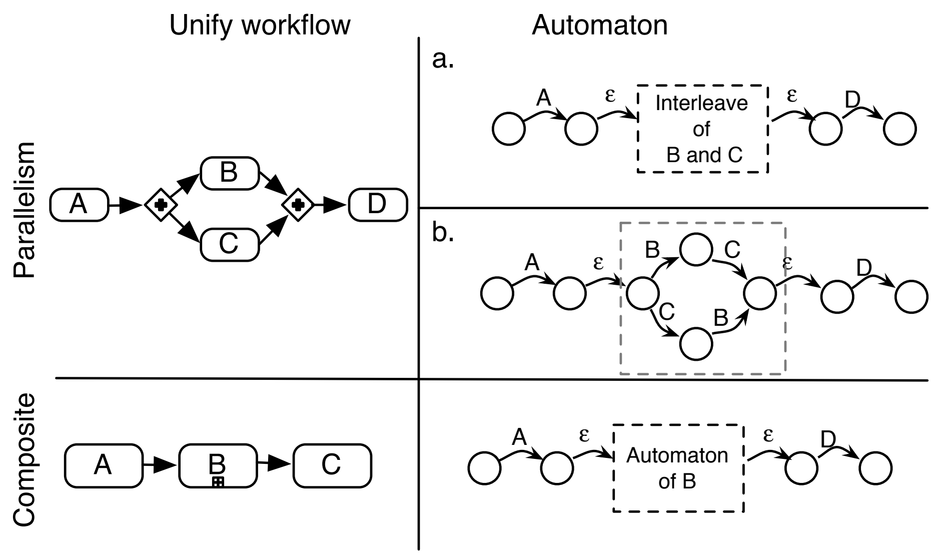


Figure 9: UNIFY workflow to automaton translation for parallelism and composite patterns

Since the translation of sub-workflows that use AndSplit and AndJoins is performed by interleaving the automaton for each branch, each branch needs to be independent from the others. Because of this restriction, the translation only succeeds if the original workflow is structured; *i.e.*, if each of the parallel branches can be enclosed using a composite activity.

Composite Activities Composite activities are handled recursively as a new workflow. Once the automaton for the composite activity is obtained, its start and end states are connected to the outer automaton using ε-transitions.

5.1.2 Implementation

The translation is implemented by means of a visitor on the AST nodes of a workflow. Upon each visit, transitions, states, or both are added to a current automaton. A stack is used to keep track of the recursive translation of the composite activities.

We have developed an automata library that provides a data structure to represent the automaton, as well as basic operations such as reduction, intersection and interleaving. It also provides translations from NFA with ε-transitions to regular NFA and to DFA. The translation is used to convert the NFA resulting from the translation of a workflow into a DFA.

In our automata library, we follow the automata and the algorithms described by [9], with one notable generalization. In literature, finite automata have a standard form where each transition has a symbol from a fixed set of symbols as its label (or the special label ε). An automaton will only consider a transition when *exactly* its symbol occurs in the input. We generalize this model, and label the transitions with a *symbol set*.

Symbol sets A symbol set consists of an ordinary set of symbols (which we will call S or T), together with a sign that may either be positive (which is the default and therefore unrepresented) or negative (represented with a "¬" prefix). In our model, the automaton will consider a transition with a symbol set when the input symbol *appears* in the set of symbols (in case of a *positive* sign), or it *does not appear* in the set of symbols (in case of a *negative* sign). For example, an input symbol B may trigger transitions labeled with symbol set $\{A, B\}$ or with symbol set $\neg\{A, C\}$, but will not trigger transitions with symbol set $\{A, C\}$ or $\neg\{A, B\}$.

Symbol sets are more expressive than singular symbols (a singular symbol A corresponds to the symbol set $\{A\}$). They prove useful in the translation of regular expressions to finite state automata, since they correspond to the standard regular expression concept of a *symbol class* (which is frequently used in practice, but which is not considered by [9]).

We may now observe that this simple structure of symbol sets is closed with respect to standard set operations such as union, intersection, and complement: the laws in Figure 10 illustrate that the result of these operations on one or two symbol sets will again be a symbol set. For example, the union of $\{A, B\}$ and $\neg\{A, C\}$ is the symbol set $\neg\{C\}$, as may be verified using standard set laws. Thanks to the algebraic laws from Figure 10, we can provide all the operations of our library (including conversion, reduction, intersection, ...) for automata labeled with symbol sets instead of singular symbols.

Union	**Intersection**	**Complement**
$S \cup T = (S \cup T)$	$S \cap T = (S \cap T)$	$\neg\ S = \neg S$
$S \cup \neg T = \neg(T \setminus S)$	$S \cap \neg T = (S \setminus T)$	$\neg\ \neg S = S$
$\neg S \cup \neg T = \neg(S \cap T)$	$\neg S \cap \neg T = \neg(S \cup T)$	

Figure 10: Algebraic laws for symbol sets

5.2 Policy to Regular Expression

We have decided to implement the predicates described in Section 4 through regular expressions. In literature, more expressive formalisms to reason about streams of activities in workflows are commonly used, Linear Temporal Logic being the most common. We have instead opted for regular expressions since they are commonly found in programming languages and utilities, therefore we believe them to be easier to grasp for developers without a background in logic.

5.2.1 Translation from Policy Predicates to Regular Expressions

Table 2 contains the predicates we have defined for expressing control flow policies in workflows. The predicates are divided into two groups.

1. The first group defines whether a given regular expression represents an example or a counterexample. The `includes` predicate is `TRUE` if the regular expression matches the workflow; while the `excludes` predicate is `TRUE` if it does not. Predicates that use `include` express existential properties of the workflow, while those that use `excludes` express universal properties.

2. The second group contains primitive predicates that are directly translated into regular expressions that must be included or excluded from the workflow. Recall that `follows(A,B)` states that the activity `A` must always be immediately followed by `B`, and a counterexample would be a workflow where `A` is succeeded by another activity, or `A` is the final activity. `between(A,B,C)` states that between the execution of `A` and `B`, a `C` must be executed, and a counterexample would be that `A` is followed by `B` with no instance of `C` in between. `may(A,B)` and `must(A,B)` respectively state that `A` may or must be followed by `B`. The former is proved by an example that follows `A` with `B`, while the latter can be disproved with a workflow execution where `A` is followed by non-`B` activities until its completion. Finally, `guards(A,B)` is disproved by a workflow execution where activity `A` occurs, and it is not preceded by a `B`.

Predicate	Definition
`includes(P)`	`TRUE` if workflow matches `P`
`excludes(P)`	`TRUE` if workflow does not match `P`
`follows(A,B)`	`excludes(.*A[^B].*|.*A)`
`between(A,B,C)`	`excludes(.*A[^C]*B.*)`
`may(A,B)`	`includes(.*A.*B.*)`
`must(A,B)`	`excludes(.*A[^B]*)`
`guards(A,B)`	`excludes([^B]*A.*)`

Table 2: Predicates and their corresponding regular expressions. `P` is a regular expression, `A,B,M` are activities.

5.2.2 Implementation

In order to evaluate the predicates that are used in the definition of the workflow policies, we must first convert the policies to regular expressions, and then translate the regular expressions into their equivalent DFAs. For the first part, we have developed a simple class hierarchy that represents each of the predicates and their corresponding regular expressions as per Table 2. The class hierarchy is shown in Figure 11. We define an AST that represents symbols, symbol classes, sequences, alternatives, and repetition (Kleene star and plus). We also identify a special atom dot (.) that stands for any symbol. As with the translation from workflows to DFAs, the DFA that corresponds to a given regular expression is obtained by traversing the AST of a given regular expression. The algorithm to do this is well known, and we have based our implementation on the one described in [9].

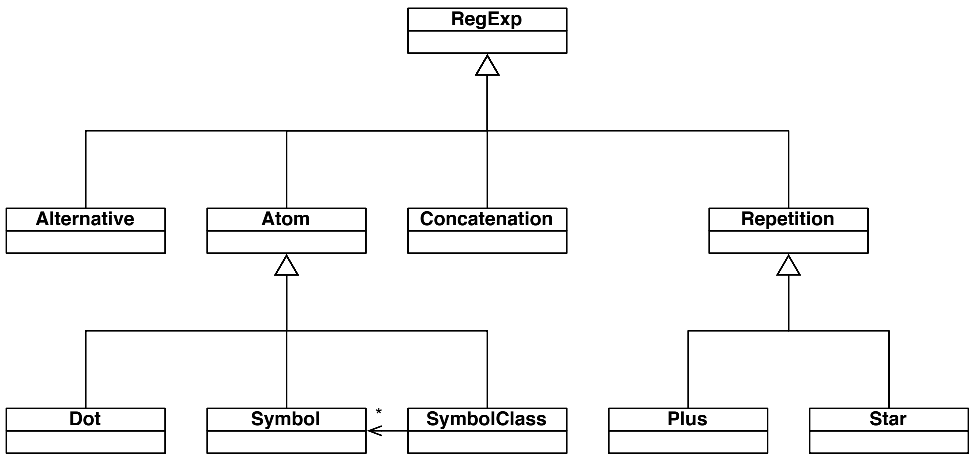
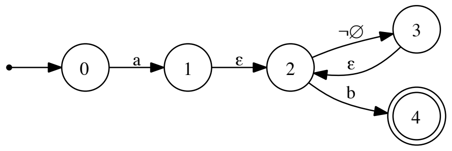


Figure 11: Class diagram for the regular expression AST

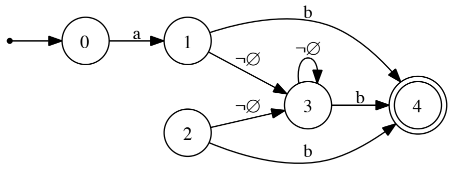
Each node of the regular expression AST is translated into the same data structure for defining automata that we used for the translation of workflows. In order to express and manipulate symbol classes and the dot meta-symbol, we may now rely on the support for symbol sets in our library, as described in the previous subsection. The use of symbol sets permit us to define and evaluate negated symbol classes[1] by representing them as the complement of the symbol set of the non-negated symbol class. Similarly, the dot meta-symbol is translated into the universal symbol set, which in turn is represented as the complement of the empty set, *i.e.*, $\neg\emptyset$.

[1]Symbol classes of the form `[^E]`, which stand for everything that is not `E`

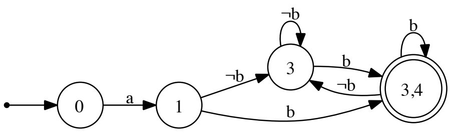
Example To illustrate in more detail how the translation from regular expression to DFA is performed, consider as an example the regular expression `a.*b`. This regular expression is first translated into the NFA with ε transitions shown in figure 12(a). In the figure, curly braces are omitted for singleton sets in the transition labels. Notice the two translations labeled $\neg\emptyset$; this stands for the complement of the empty symbol set, *i.e.*, the universal symbol set. In figure 12(b) and 12(c), the equivalent NFA and DFA are depicted, respectively.



(a) NFA with ε-transitions



(b) NFA



(c) Final DFA

Figure 12: Steps for the translation of the regular expressions into a DFA

5.3 Intersection of DFAs

Now that we have defined how to obtain a DFA for both the workflow and the policy, the final step in the verification consists in the construction of the intersection of both DFAs. The intersection of two DFAs is another DFA that accepts a language that is exactly the intersection of the languages of each of the individual DFAs. If this intersection automaton still has satisfiable paths (*i.e.*, if its language is not empty), then this path represents a possible execution of the workflow that is relevant to the policy (because it is either an example or a counterexample).

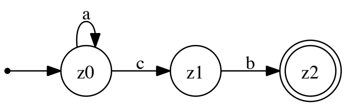
5.3.1 Construction of the Intersection DFA

As explained by [9], it is possible to directly construct an automaton M that intersects two DFAs M_1 and M_2, although we have to update their algorithm to account for the use of symbol sets instead of singular symbols. The intersection automaton is again a DFA, which is constructed as follows:

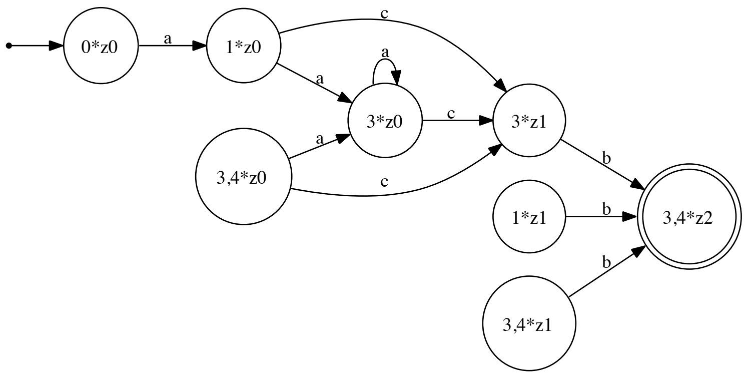
- The set of states of M is the product of the states of M_1 and M_2. In other words, each state of M is of the form $[p_1, p_2]$ where p_1 is a state of M_1 and p_2 of M_2.
- The state $[p_1, p_2]$ is final in M, if both p_1 is final in M_1, and p_2 is final in M_2.
- The starting state of M is $[q_1, q_2]$, where q_1 is the starting state of M_1, and q_2 is the starting state of M_2.

- For each transition from p_1 with symbol set a to p'_1 in M_1, and each transition from p_2 with symbol set b to p'_2 in M_2, we add a transition from $[p_1, p_2]$ with symbol set $a \cap b$ to $[p'_1, p'_2]$ in M.

Example As an example, Figure 13(b) shows the result of the intersection of the automaton from Figure 12(c) with a new automaton, which is shown in Figure 13(a).[2]



(a) Example automaton



(b) Resulting intersection

Figure 13: Example of the intersection of the given automaton and the automaton from Figure 12(c)

5.3.2 Analysis of Satisfiable Paths

The goal of this analysis is to determine whether the language of the resulting automaton is non-empty, *i.e.*, whether there exists a path from the initial state to a final state. This may be done by using an iterative coloring of the states of the automaton, to mark which states are reachable.

Initially, we color only the starting state. At each iteration we follow the outgoing transitions of newly colored states to color states that we have not considered before (we only follow transitions if they have a non-empty symbol set). Since the number of colored states grows at every iteration, and since there are only a finite number of states, this procedure will terminate at some point. At that point, we have found all the states that are reachable in the automaton. If this set includes a final state, the language of the automaton is non-empty.

[2]States that do not have any incoming or outgoing transitions have been removed.

6 Conclusions

In this technical report, we present a language to express workflow policies for verifying control flow interactions in workflows in the presence of aspects, as well as a technique based on deterministic state automata (DFA) to verify these policies.

Inspired by previous work [5], we propose a predicate logic language for expressing workflow policies. Predicates in workflow policies reason about the ordering of the execution of the activities present in a workflow. We base our approach on representing workflows as DFAs, translate policies to regular expressions and then match the regular expressions to the workflows' DFAs. This process is implemented for the UNIFY [11] workflow system.

The inclusion of ordering information in our approach presents an improvement over [5], which represented traces of Java programs as unordered sets. Additionally, control flow in workflows is more complex than control flow in [5] due to workflows' explicit parallelism constructs. The use of DFAs in our approach to represent workflows gives an accurate representation of the behavior of the workflows and takes into account parallel constructs by including all possible interleavings of parallel activities. The implementation of the predicates as regular expressions allows for a straightforward implementation of the checking of the predicates (DFA intersection), as well as opening the door to user-defined predicates, since regular expressions are familiar constructs to developers.

While inspired by the problem of control flow interaction between aspects, our approach remains applicable in the more general context of workflow verification, since neither the predicates we propose nor the technique to evaluate them are specific to aspects. As possible avenues for future work, we hope to address the issues of scalability and expressiveness of workflow policies. First, industrial-sized case studies are needed to assess the scalability of our approach compared to model-checking based proposals such as the one presented by [7]. Second, regular expressions are not expressive enough to encode certain common patterns in workflows' behavior, *e.g.*, paired activities. A more expressive formalism that retains the advantages of regular expressions would be desirable.

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. 5

[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. 4, 5, 7

[3] Anis Charfi and Mira Mezini. Aspect-oriented web service composition. In Liang-Jie Zhang and Mario Jeckle, editors, *Proc. European Conf. on Web Services (ECOWS-2004)*, volume 3250 of *LNCS*, pages 168–182, Berlin, September 2004. Springer Verlag. 4, 5, 7

[4] Carine Courbis and Anthony Finkelstein. Towards aspect weaving applications. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *Proc. 27th Int' Conf. on Software Engineering (ICSE-2005)*, pages 69–77. ACM Press, 2005. 4, 5, 7

[5] Bruno De Fraine, Pablo Daniel Quiroga, and Viviane Jonckers. Management of aspect interactions using statically-verified control-flow relations. In *Proceedings of the 3rd International Workshop on Aspects, Dependencies and Interactions (ADI 2008)*, July 2008. 4, 12, 23

[6] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In Don Batory, Charles Consel, and Walid Taha, editors, *Proc. 1st ACM SIGPLAN/SIGSOFT Conf. on Generative Programming and Component Engineering (GPCE-2002)*, volume 2487 of *LNCS*, pages 173–188. Springer Verlag, October 2002. 12

[7] Alexander Förster, Gregor Engels, Tim Schattkowsky, and Ragnhild Van Der Straeten. Verification of business process quality constraints based on visual process patterns. In *Symposium on Theoretical Aspects of Software Engineering (TASE'07)*, pages 197–208. IEEE/IFIP, 2007. 23

[8] 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, 2009. Springer. 7

[9] John E. Hopcroft and Jeffry D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979. 16, 18, 19, 20, 21

[10] 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. 7

[11] 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). 7, 10, 23

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proc. 11th European Conf. on Object-Oriented Programming (ECOOP-1997)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997. 4, 5, 7

[13] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. 5

[14] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, June 2005. 5

[15] Stephen A. White. Business Process Modeling Notation, version 1.0, May 2004. 5