A flexible architecture to build workflows using aspect-oriented concepts

Mario Sánchez, Jorge Villalobos
Universidad de los Andes
Bogotá, Colombia
{mar-san1,jvillalo}@uniandes.edu.co

ABSTRACT

Since many workflow applications are used in contexts where the requirements and business rules change frequently, it is necessary to build those applications using strategies and tools that favor adaptation and reuse. The goal of this paper is to show an approach to build these extensible workflow applications using synchronized executable models. This approach uses concepts related to aspect-oriented software development, such as concern separation and instrumentation; thus, in addition to presenting the approach, we discuss our view on the central characteristics that define aspect-modeling, and we show how these concepts relate to our work and how they can be applied to workflow applications.

Keywords

Workflow domains, Aspect-oriented modeling, Executable models

1. INTRODUCTION

Aspect-oriented modeling is a technique that is frequently used to guide the design and construction of many different kinds of applications. Starting from the original proposal of Aspect-oriented programming of Kiczales et al. [5], aspects have been adopted in many languages and contexts. Since not all the implementations are exactly alike, this opens a discussion about what elements are fundamental to the aspects proposal. In this paper we pose our view on what elements are common to every aspect-oriented modeling technique, and we illustrate this in the context of the construction of workflow based applications.

Today, workflows are increasingly being used and this has led to the definition of several languages and applications to define and enact workflow processes. Moreover, there are several applications that rely on workflow engines to support some parts of its semantics. A common element in these tools is a central coordination element that leads the cooperative execution of several active entities, and provides a way to integrate them to achieve a common goal [1]. This central element is usually called the control dimension: it comprises and executes entities that are related to the control-flow, that is, elements that describe tasks and their execution ordering [10] to form processes. As an example, a BPEL engine controls the access to web-services and thus composes them to create more complex services. Another example are applications that control the interaction between humans and machines by routing and transforming the data produced in each step.

Currently, the problem is that workflows are mostly used in contexts where there are frequent changes of requirements and business rules; moreover, there is also a constant need for new applications, and the particular requirements in each case make it difficult to reuse existing applications. Due to this, it is necessary to find strategies and tools to easily build and adapt workflow applications. This can be achieved by providing mechanisms to modularize, describe, adapt, reuse and compose parts of the solutions.

The goal of this paper is to present our approach to build workflow applications, and show how it relies on several concepts that are basic to aspect-modeling. The approach that we have developed is based on the usage of synchronized executable models to represent different concerns that participate in a workflow application. This is different from many aspect-based strategies in that weaving is done over models, and the result is not a single executable element but a set of synchronized executable models. However, this approach has several similarities to general aspect concepts, and offers all of the advantages typically associated to aspect-modeling.

The rest of the paper is organized as follows. Section 2 discusses some approaches to model workflows and the possibility of identifying domains within them. Next, section 3 presents our view on the concepts that are fundamental to aspect-modeling. Section 4 relates the ideas presented in section 2 and 3 by showing our approach to implement workflows using executable models in an aspect-like way. In section 5 we describe one sample application, and in section 6 we present the conclusions of the paper.

2. WORKFLOW DOMAINS

The goal of this section is to show that workflows are indeed a kind of applications that can greatly benefit from the usage of aspects *independently* from the technologies used to implement them.

Several languages and tools to model and enact workflows have appeared in recent years. Each one of those languages usually focused on modeling specific kind of processes or offering a specific advantage over other alternatives. For instance, BPEL was designed to compose web-services and thus it requires low-level descriptions of the processes and of the interaction between services and data; on the other hand, BPMN is mainly used to document processes from a business perspective and offers high-level descriptions, but it is not directly executable; a third alternative is YAWL, that was designed as an intermediate language and has a formal specification of its execution semantics. Other well known languages and tools are XPDL, an interchangeable format to describe processes, and the Microsoft Workflow Foundation, that offers a framework with base elements to build workflows.

When a process is fully defined to be enacted it should include several small, extra elements and tasks that complement or support the main activities, but are not related to the business process. These elements are usually recurrent and they can address a wide variety of details. For example, some of them serve to assign resources to the main activities; others serve to store, retrieve and transform data; and others serve to check the compliance to certain restrictions, such as deadlines and time-outs. Because of this wide range of possibilities, they also vary in their structure and can have a great impact on the structure of the process. For instance, specifying the maximum duration of an activity can impact the structure of a process beyond that single activity (see [10], pag. 20, figure 7, for some examples of how introducing timing affects the process' structure).

These extra elements and tasks can be transparently added by the supporting tools, or they can be explicitly included in the process definition. This is achieved with the help of special constructs of the language used to describe the process. For instance, BPEL offers constructs to handle time-outs. A direct consequence of this is that the definition of the process becomes entangled with the definition of elements that are not part of the domain where the workflow is used; additionally, it also forces the designer of the process to take into account these extra things. Another disadvantage of this situation is that it hinders evolution of the application because of the high coupling between all its elements.

A possible strategy to solve this problem is to decouple the process definition from the definition of the supplemental tasks and elements. These separated elements are then grouped in domains, which serve to relate elements that address similar goals. For instance, the domain of resources groups the elements used to define the available resources, and the rules to assign them to activities. A very special domain is the control domain that includes the elements used to define and structure the tasks that are performed in a workflow. After identifying the domains and their elements, the other important step in this strategy is to identify the relationships between elements from different domains. This separation of domains favors the comprehension of the process and the evolution of its parts. However, there is still the problem of how to describe and implement the decoupled elements and their relationships.

Several works have addressed parts of this problem. For example, in [8] several domains are first identified and then they focus on the analysis of the data domain. There are also tools that implement similar ideas, such as AO4BPEL [3] and Padus [2], two existing aspect-oriented extensions to BPEL. AO4BPEL uses an XPath based pointcut language to weave advices over existing BPEL processes and the resulting processes are executed in a modified BPEL engine. AO4BPEL has an important disadvantage in the fact that advices have to be modeled using BPEL. On the other hand, Padus is based in advices defined using BPEL, and pointcuts described with a logic language. Since it uses a static weaving system that outputs BPEL processes, it is possible to execute the weaved applications in well-known and very efficient standard BPEL engines. However, Padus suffers from the same limitations of AO4BPEL because of the usage of BPEL as advice language: the things that can be said in the advices are limited to what can be normally be said in BPEL, even if the concerns modeled do not have anything to do with control. The approach that we present in section 4 of this paper makes it possible to use different definition languages for different concerns.

3. ASPECT-ORIENTED CONCEPTS

Starting with the original works of Kiczales et al. [5], discussions have been risen to identify issues such as the limitations and advantages of aspects, their possible fields of applications, and the concepts that are central to the orientation. In [6], Filman and Friedman proposed that the distinguishing characteristics of aspect-oriented programming and modeling are *quantification* and *obliviousness*; this work served to Steimann as a basis for [9], starting a discussion with the authors of [7], which proposed *abstraction*, *modularity* and *composability* as fundamental properties of AOP. We realize that these works had as main focus to obtain and discuss a definition of aspect-oriented programming, whereas our work's main focus is on workflow applications. We do not expect to confirm or refute any of these previous works, so we present a small set of aspect-oriented concepts that we have identified with our experience and serve to contextualize our work.

The first of these concepts is the separation of crosscutting concerns, intended as the identification and isolation of several aspects of the system. The second concept is instrumentation, intended as the mechanism used to materialize the relationships between concerns that are required to have fully working applications. These two concepts always appear in aspect implementations, are consistent with both the definitions of [7] and [6], and they are independent of the languages and tools used. Furthermore, these two concepts can be applied and bring benefits to applications in many different contexts. In the rest of the section we will further describe these two concepts. Since we will illustrate them using code-oriented implementations of aspect technologies, we will use terms such as base-program and code instrumentation.

Separation of crosscutting concerns

As presented in [5], the proposal of separating crosscutting concerns comes from the idea that using only one abstraction can be insufficient to handle all the issues that appear in an application. Also in [5] it is said that "different aspects of a systems behavior that must be programmed, each tend to have their own 'natural form', so while one abstraction framework might do a good job of capturing one aspect, it will do a less good job capturing others". From this point of view, separation of concerns allows the usage of models and languages with the adequate expressiveness for each case.

The separation of concerns is not always done *a-priori* in an aspect-based application. In many cases it is possible to identify new concerns very late in the development cycle. In these cases, the aspect-orientation becomes a facilitator for unexpected extension, and applications can then be extended in ways that were unexpected when they were initially developed. How profound are the extensions depends on the expressiveness of the models and languages used to describe those extensions. Another factor here is that concerns can be separated in several ways even in a single application. For instance, in java applications it is common to group the main functionalities in a concern that is called the base program, and in several other concerns that handle extra requirements such as logging or transaction management. However, it is important that concerns must be completely separated from the base code and also between them: they should not know about each other prior to weaving, and it should be possible to execute the base program even when one or all of the crosscutting concerns are missing.

Another characteristic of concern separation is that it should be possible to locate composition and coordination hooks in the base program (join points) and in each concern (advices). The hooks available in the base program depend on the technology used. In the case of java applications, joint points are usually associated to method calls. There are also some recent tools that allow the location of specific lines of code. In all cases, the finer the granularity of the join points, the more expressive the weaving strategy becomes. Adversely, the elements described in a concern are grouped in 'advices', which are located by simply using a name. The capacity of locating elements in the base program and in the concerns makes it possible to design pointcut languages, which are fundamental to the weaving process.

Instrumentation

The other essential concept of the aspect-orientation is the usage of instrumentation to weave the concerns. Using this mechanism, it is possible to take the execution control from one concern and give it temporarily to another; in the case of code-based applications, this means that instrumentation is used to take the control from the base program and execute some other code defined in a concern.

Instrumentation is also a mechanism to do white-box composition between concerns. This means that it is necessary to know the internal structure of these elements in order to establish relationships between them; if they were taken as black-boxes, it would be impossible to locate arbitrary join points and the possibilities of extension would only be the ones foreseen by the initial developers. In the case of object-oriented programming, this means that extensibility would be limited to the interfaces provided, thus eliminating the possibility of unexpected extensions.

Depending on the technologies involved, several weaving strategies can be used, and each one offers a different trade-off. For instance, static weavers that pre-process the source files before compilation can use standard compilers, but have a great impact on the code that is executed because it is difficult to distinguish the base-code from the aspect-code. On the other hand, a coordination-based weaver, like the one that will be presented in section 4, can maintain a clear separation between the original codes. Other weaving strategies exist and depend on things like class inheritance, decorators, proxies or byte-code modification.

In order to use an instrumentation strategy to weave a base program with its concerns, it is necessary to have some information about the relationships that have to be materialized. These links relate join points of the base program (i.e. methods) with advices using a specific instrumentation strategy. This information is described using a pointcut language that is developed for each aspect technology.

This concludes our discussion about the base concepts behind aspect-oriented modeling. In it, we identified one point that relates to modularization, and one point that relates to the composition of the modules obtained.

4. MATERIALIZING CONCERNS AS EXECUTABLE MODELS

In this section we will present our approach to build workflow applications. This approach is based on the idea of identifying domains involved in a workflow and implementing them as concerns using executable models. The relationships identified between domains are materialized using a weaving strategy that synchronizes the executable models using a coordination mechanism based on event passing and method calls. All our executable models are constructed using a coordination element that we call open object [11]: it provides a way to represent the state of the elements appearing in a model, and offers all the necessary synchronization mechanisms. We will now present in more detail the structure and characteristics of an open object, and then we will show how they can be extended and grouped to represent specific concerns.

An open object is composed of an entity, a state machine associated to the entity, and a set of actions. An *entity* is just a traditional object with attributes and methods. It provides an attribute-based state to the open object and its methods are a place where part of its behavior can be implemented. The *state machine* materializes an abstraction of the life-cycle of the entity, allowing other elements to know this state and react to its changes. Finally, the *actions* are pieces of behavior that are associated to transitions of the state machine. When a transition is taken, its actions are executed in a synchronized way.

To specify a state machine it is necessary to describe its states, the transitions between states and the events that will trigger each transition. For instance, to keep the state machine consistent with the internal-state of the entity, each time the latter is modified, it generates an event. The state machine receives that event, processes it, and takes the transition associated to that particular event. When a state machine changes its state, this also generates an event. This mechanism can also be used to react to changes in other open objects. Moreover, open objects also offer a synchronization mechanism that is not based on events. When a transition is taken, all the actions associated to that transition are executed, and these actions can make direct calls to other elements to invoke their methods.

Figure 1 shows a simplified version of an open object that represents an activity. There is an entity, (the class Activity itself), a state machine, and one action associated to one of the transitions in the state machine. The state machine reacts to events that are generated by entity methods and are related to the transitions. For instance, the method `activate( )` generates an event that takes the state machine from `Inactive` to `Active`. On the other hand, the event generated by the method `finish( )` changes the state of the

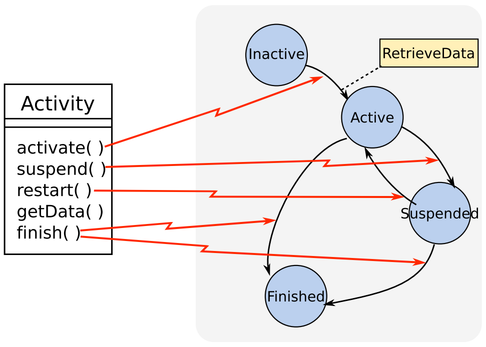

Figure 1: Sample open object: Activity.

state machine from `Active` to `Finished` or from `Suspended` to `Finished`. When the transition between `Inactive` and `Active` is taken, the action `RetrieveData` is executed: the behavior implemented in this action is invoking the method `getData( )` of the activity.

To support the flexibility requirements of workflows, and to allow elements to be adapted individually, the open objects have three different mechanisms to alter or extend their structure and behavior. The simplest mechanism involves only the modification of the actions of an open object. This mechanism is used to enrich the semantics of the transitions and is very useful when building workflow applications and synchronizing concerns. The second extension mechanisms involves the redefinition of the state machine of an open object; this mechanism has a richer expressiveness than the previous one. Finally, the third mechanism involves the modification of the entity of an open object; this is achieved with a total replacement of the associated object, or with the creation of an extended version (possibly using inheritance). These three mechanisms are fundamental for the creation of models to represent particular concerns.

In our approach, each concern of a workflow is represented by an executable model based on open objects. Furthermore, to build each concern there is a different set of open objects available, that we call a metamodel. These metamodels can be used as a template to build models, and their elements are not totally fixed and can be extended and customized. This is similar to the strategy used in [12], but instead on relying on MOF and classes, we use open objects. As an example, we have a metamodel called XTM (eXtensible Time Modeling) that we use to build models representing the time concern of any workflow. The creation of XTM required the definition of several open objects that represent the basic elements of the concern. Whenever a new time model is created using XTM, the available open objects are adapted and extended as necessary to support the specific requirements of the specific process.

The most important metamodel that we have defined is called XPM (eXtensible Process Modeling), and it serves to model the elements that are part of the control dimension of an application, that is, the set of tasks that have to be executed in a precise order to achieve a specific goal [4]. Some of the elements available in this metamodel are Process, Activity, Multiactivity, Port and Dataflow; in [11] there is a complete description of these elements, as well as an example of a complete process. All the elements of the metamodel are described as open obejcts, and each one has a state machine that abstracts its life-cycle (see figure 1). The structure of these state machines is fundamental to describe the weaving between concerns.

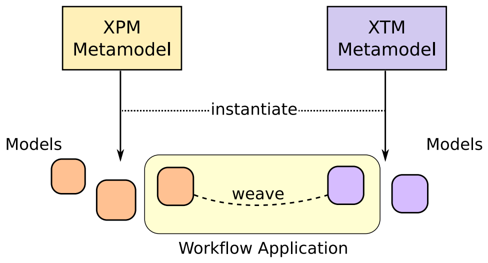

Figure 2: Models, Metamodels, and Workflow Applications.

As figure 2 shows, each metamodel is used to create models, and these models together with some linkages forms workflow applications: each model represents one of the concerns involved in an application, and the links between them are responsible for their coordination. This is achieved by using a weaving strategy that coordinates the open objects to synchronize the various executable models. In this weaving strategy, to synchronize two models it is fist necessary to locate a transition that will be used as join point. Next, an open object that is interested in being notified about the transition is identified in the other model. Finally, it is necessary to add an action to the transition and to configure it to notify the element identified in the second model. This notification is usually performed by invoking a method of the entity.

The weaving strategy presented can be considered an instrumentation based strategy. By adding an action, the execution control is synchronically transfered to the other concern. The behavior of the original model is not altered and thus if the weaved concern does not performs any operation, then the behavior of the application remains unchanged.

The responsible of performing the weaving is a component that is separated from all the models. This weaver is capable of interpreting a pointcut language that serves to define all the information required, including the type of action used to establish the link.

We will now illustrate these ideas using a fragment of a sample workflow. This application includes a process (domain of control), and time restrictions that apply to the execution of the process (domain of time). To implement this workflow two models are defined, one for each concern. In the rest of the explanation, we will use the activity in the process that is called A, and the time restriction called TR1 that specifies that A's execution should last less than 2 hours. The state machines of these elements are shown in figure 3. In this image we have also included a time counter that checks the time for TR1. This time counter is another open object, with methods to start and stop that are called by TR1's actions, but for simplicity we have represented it as a small clock.

In order to synchronize the two open objects A and TR1, it is necessary to establish relationships between them: the time restriction TR1 should start checking the time (using the time counter started by the action `StartTC`) as soon as

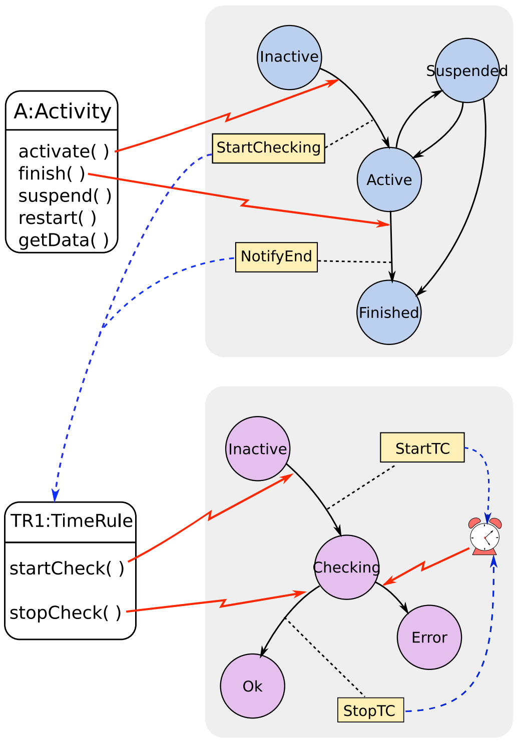

Figure 3: Relations between an activity and a time restriction.

activity A starts its execution; furthermore, the time restriction TR1 should be notified when activity A finishes its execution and stop the time counter. These relationships are created by the weaver using instructions such as the following:

```
process:(Activity)A|activation
StartChecking (
time:(TimeRestriction)TR1,
startCheck( ) )

process:(Activity)A|deactivation
NotifyEnd (
time:(TimeRestriction)TR1,
stopCheck( ))
```

As said previously, the weaving mechanism is based on adding actions. The first block of instructions specifies that the action `StartChecking` should be associated to the transition called **activation** of activity A, and that this action should invoke the method `startCheck( )` of the time restriction TR1. The second block of instructions specifies that the action `NotifyEnd` should be associated to the transition called **deactivation** of activity A, and that this action should invoke the method `stopCheck( )` of the time restriction TR1. Other similar instructions are used to relate the state of TR1 with the transitions that go to and from the state `Suspended` of A.

In this example we have shown how to weave one specific activity to one specific time rule. However, it is also possible to weave one time rule to several activities or to specify that each activity has to be woven to a time rule.

5. EXPERIMENTATION

The approach that we have presented in this paper has been implemented, tested and used to build several workflow-based applications. One of these applications is called PaperXpress and it is a tool to support collaborative writing of research papers. In PaperXpress, ad-hoc processes are defined and then the tool offers the support needed to enact the processes by coordinating its tasks and handling the results of the writing process.

When PaperXpress is executed, there are three domains involved: a control model, a resource model and a data model. The control model describes the activities that have to be executed in order to write the paper; the resource model describes who is responsible for each of the tasks; and the data model holds all the relevant information, including the structure and the contents of the paper that is being written.

In this application, the metamodels of control and resources are extensions to metamodels that we had already used, such as XPM. Only the metamodel of data was developed from scratch for this application. Because of this, PaperXpress is also an example of how the approach presented impacts the development cycle of an application by favoring reuse.

The following is a small list of the steps required to build similar applications, with a special emphasis on the reuse of metamodels.

1. Identify the domains that are going to be part of the application.

2. Study the available metamodels, created for previous projects, and select those that can be used to model the domains identified in step 1.

3. Adapt and extend the metamodels that require adjustments in order to be used in the new application.

4. Build new metamodels for the remaining dimensions.

5. Build the rest of the application, using the selected metamodels to represent the concerns involved.

Besides showing the advantages of the approach that are related to reuse, this also shows important benefits related to evolution. Specifically, that the metamodels can evolve independently and it is even possible to add new concerns. Furthermore, the relationships between concerns are totally decoupled and they can also evolve independently.

6. CONCLUSIONS

In this paper we have presented our approach to build workflow applications and we have motivated some reasons to consider it an aspect-oriented approach. First, we identified some characteristics of workflow applications that makes them suitable to be modeled from an aspect-oriented perspective. Then, we discussed what we consider the basic concepts behind the aspects orientation: separation of concerns and instrumentation. These concepts are rather evident in code-oriented aspect implementations, but they can also be applied in other kind of applications.

We propose the idea that workflow applications can be constructed using executable models which are synchronized with method calls and event passing. This provides an aspect framework where the concerns are represented with executable models, and the weaving is based on a coordination mechanism.

The usage of aspects to build workflow applications is interesting form several points of view. It facilitates the usage of different models to represent each dimension that is involved in the workflow. It also makes extensions simpler to create because the workflow descriptions become less entangled. Finally, the approach presented also offers advantages related to reuse, since it identifies some specific reusable artifacts and offers mechanisms to adapt these artifacts to new requirements.

7. ACKNOWLEDGMENTS

This work is supported by the VLIR funded CARAMELOS project (http://ssel.vub.ac.be/caramelos/) and by Colciencias (http://www.colciencias.gov.co).

8. REFERENCES

[1] Papadopoulos, G. A. and F. Arbab, Coordination models and languages. In 761, Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 1998 p. 55.

[2] Braem, M. et al. Isolating Process-Level Concerns Using Padus. In Proceedings of the 4th International Conference on Business Process Management (BPM 2006), Vienna, Austria, September 2006. LNCS Springer-Verlag.

[3] Charfi, A., Mezini, M., Aspect-Oriented Workflow Languages. In OTM Confederated International Conferences, Montpellier, France, 2006, Lecture Notes in Computer Science 4275, pp. 183-200.

[4] Craven, N., Mahling, D.: Goals and processes: a task basis for pro jects and workflows. In: COCS 95: Proceedings of conference on Organizational computing systems, New York, NY, USA, ACM Press (1995) 237-248

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In ECOOP'97–Object-Oriented Programming, 11th European Conference, LNCS 1241, pages 220-242, 1997.

[6] R. Filman, D. Friedman, Aspect-Oriented Programming is Quantification and Obliviousness. In OOPSLA WS on Advanced Separation of Concerns, 2000.

[7] A. Rashid, A. Moreira, Domain Models are NOT Aspect Free. In MoDELS 2006, Springer, LNCS 4199, pp. 155-169.

[8] Sadiq, S., M. Orlowska, W. Sadiq and C. Foulger, Data flow and validation in workflow modelling. In ADC 04: Proceedings of the 15th Australasian database conference (2004), pp. 207-214

[9] Steimann, F., Domain models are aspect free. In MoDELS 2005, 8th International Conference on Model Driven Engineering Languages and Systems (2005) 171-185.

[10] van der Aalst, W., L. Aldred, M. Dumas and A. ter Hofstede, Design and implementation of the yawl system. In: Advanced Information Systems Engineering. 16th International Conference, CAiSE 2004, Riga, Latvia, June 2004, Lecture Notes in Computer Science 3084 (2004), pp. 291-308.

[11] Villalobos, J. Sánchez, M. and Romero, D. Executable Models as Composition Elements in the Construction of Families of Applications. 6th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2007), Portugal, September 2007.

[12] Object Management Group: Software Process Engineering Metamodel (SPEM), Version 1.1 (January 2005)