

# Supporting Evolution in Workflow Definition Languages

Sérgio Miguel Fernandes, João Cachopo, and António Rito Silva

INESC-ID/Technical University of Lisbon  
Rua Alves Redol n° 9, 1000-029 Lisboa, Portugal  
{Sergio.Fernandes, Joao.Cachopo, Rito.Silva}@inesc-id.pt  
<http://www.esw.inesc-id.pt>

**Abstract.** Workflow definition languages are evolving rapidly. However, there is not any agreed-upon standard for such languages. In this paper we address the problem of how to develop a workflow system that is able to cope with the constant changes in these languages. We address the definition aspect of workflow systems by distinguishing between frontend and backend languages. This way, a workflow system can be developed based on the backend language whereas the frontend language can change.

## 1 Introduction

In its simplest form, a *Workflow System* (WfS) allows its users to describe and execute a workflow. The language used to describe a workflow is a *Workflow Definition Language* (WfDL).

In this paper we address a software engineering problem: The problem of developing a WfS that supports changes in the WfDL. Moreover, it should be able to support more than one WfDL simultaneously. The solution we propose is to split the WfS into two layers: (1) a layer implementing a *Workflow Virtual Machine*, which is responsible for most of the WfS activities; and (2) a layer where the different WfDLs are handled, which is responsible for making the mapping between each WfDL and the Workflow Virtual Machine.

In the next section, we give some motivation for this work and discuss some of the problems that a WfS faces when it was not built with WfDL change in mind. In Sect. 3 we describe our proposal of creating a Workflow Virtual Machine to support different WfDLs, and in Sect. 4 we describe the object-oriented framework we implemented using this approach. Then, in Sect. 5 we discuss some related work and open issues. Finally, in Sect. 6 we conclude the paper.

## 2 Motivation

There are many WfSs available, each one of them with its own WfDL—different from all the others. Actually, despite the efforts of the Workflow Management

Coalition [5], different languages give different names to the same concept (e.g. some call “Activity” to what others call “Step”). The lack of a common language to describe workflows makes impossible the easy migration from one WfS to another. This is a problem for users—those that want to model and execute workflows.

Some authors have already recognised this problem [9] and presented a critical evaluation of some of these languages. This problem would be solved if we had a standard WfDL that all systems supported. As a matter of fact, several groups presented their proposals for such a standard [2, 3]. Yet, so far, the workflow community has not reached an agreement on which language to adopt.<sup>1</sup>

This proliferation of languages has disadvantages also for the developers of WfSs. It poses the question of which language to support in the system under development. If the chosen language becomes the standard, fine, but what if the chosen language is abandoned?

Even if a consensus was reached, the agreed upon language would be changing over time, as the specifications themselves are not yet stable, containing many open issues, as their own creators acknowledge [3, 2].

The WfS exists to support the execution of workflows, which are described by a WfDL. That WfDL has a certain set of concepts and constructs. Naturally, the WfS has to deal with those concepts and constructs to perform its tasks (such as the execution of a workflow). If the WfDL changes significantly (e.g., because it was abandoned and replaced by another), the WfS developers may have to rewrite a significant part of the system. Therefore, a major concern for the WfS developers is how to make it able to cope with changes in the WfDL, with the minimum impact in the rest of the system.

### 3 The Workflow Virtual Machine

To isolate the core of the WfS from changes to the WfDL, we propose the creation of a *Workflow Virtual Machine* (WfVM).

Although there are many workflow definition languages, they all share a common subset of concepts such as “activities,” “activity sequencing,” “data handling,” and “invoking domain-specific logic” [5]. The idea of the WfVM is that it should support these core concepts, allowing the definition and the execution of workflows described in a simple language that uses those concepts. We call this language the *backend language*. This is not the language that WfS users use to describe their workflows. That language—the WfDL we have been talking about—is what we call the *frontend language*.

With a WfVM in place, the description of a workflow in the frontend language is compiled into a workflow described in the backend language and then executed. This way, the core of the WfS does not depend on the frontend language, which can change freely—provided that it can always be compiled to an appropriate backend description.

---

<sup>1</sup> Unfortunately, we cannot foresee in a near future such an agreement.

This separation between frontend and backend languages has two additional benefits. First, we can have different frontend languages simultaneously very easily. A universal WfDL that suits all users is not feasible. For particular domains, some constructs related to the domain can simplify greatly the modelling task. Of course, different domains have different needs. So, being able to support different frontend languages is an important requirement for a successful WfS.

Second, the existence of two distinct languages solves the conflict between the two forces that guide the specification of a WfDL: From the standpoint of a WfS user, the language should be easy to use (i.e., user-friendly), preferably supported by modelling tools, containing many constructs to help on the modelling process, and designed to help the user to avoid making errors. On the other hand, we have the WfS developers, who are concerned with qualities such as extensibility, maintainability and simplicity of implementation—therefore, the language should be minimal, with no redundancy in it. The backend language is not to be used by humans, so it does not have the user-friendliness requirements; it can meet the requirements of the WfS developers. The language to be used by humans (the frontend language), is not to be operated by the WfS (apart from the compilation process); it can be as large and user-friendly as we want.

To conclude this section, note that there are some important requirements for the backend language: it needs to be expressive enough to support different frontend languages, and it needs to be flexible enough to adapt smoothly to new concepts that may be expressed in future frontend languages. We will not examine in detail these requirements in the paper but we will discuss them briefly in Sect. 5.

## 4 Workflow Framework Implementation

We developed an object-oriented workflow framework using the approach proposed in this paper: The core of the framework implements a Workflow Virtual Machine, supporting the execution of workflows defined in a backend language we designed. The framework also supports a frontend language—based on the work of Manolescu [7]—which is compiled to the backend language. Due to space limitations, we cannot describe the framework in detail but we will discuss some of its aspects to illustrate our approach.

The core of the framework is composed of two modules: the Definition Module, and the Execution Module. The former is responsible for managing workflow definitions whereas the latter is responsible for managing the execution of instances of the stored definitions.

In the next section, we present the backend language, and in Sect. 4.2 we describe how a workflow is executed. This gives an operational semantic to the backend language. Next, in Sect. 4.3 we present the frontend language we referred to above. Finally, Sect. 4.4 shows a mapping from the frontend language to the backend language.

## 4.1 Backend Definition

The basic structure for the backend workflow definition is a directed graph. In the graph, the nodes represent activity steps, and the transitions between nodes represent the flow. The flow can be either control or data. Control flow establishes the order in which the graph should be navigated to execute its nodes. Data flow defines what data is passed from one activity to another. In this paper we will focus on the control flow aspects only.

Every graph should have exactly one start node and one end node. A node represents a *step*. Every step has a *service object* associated which contains the node's domain-specific behaviour. The service object is invoked every time the step is scheduled for execution by the engine. When this invocation returns, the engine assumes that the domain-specific operations for this step have been completed and graph navigation can proceed.<sup>2</sup>

*Control flows* link steps. A control flow is a directed edge in a graph from one node to another. The direction specifies the order in which the two steps will be performed. The source step must have ended before the target begins execution. Every control flow has a *transition condition*, which indicates whether the flow of control can proceed through that transition to the target step.

*Concurrency* is supported by having more than one control flow originating from the same step, thus creating two or more parallel flows. Concurrent work can be *synchronised* later by having more than one control flow targeting the same step. Steps have *synchronisation rules*. Currently, there are two rules available: the *OR-rule* and the *AND-rule*. A step that has an OR synchronisation rule will be scheduled for execution when the first control flow reaches that step (a control flow reaches a step when a preceding step ends and the transition condition of the connecting control flow evaluates to true). When the following control flows reach the same step, they are ignored. A join step that has an AND synchronisation rule will be scheduled for execution when all preceding control flows reach that step.

## 4.2 Execution Module

The execution module contains the life-cycle-management operations for the workflow instances. The execution module is responsible for reading the shared workflow definitions from the definition module and for managing each workflow instance in a separate execution context. This way, the framework can concurrently execute multiple workflows sharing the same definition. For each *execute* operation, the engine creates an object—the *executor*—responsible for holding all the information relevant for executing a single workflow instance, as well as carrying out the graph navigation algorithm. The following summarises the executor algorithm:

1. The execution starts on the graph's start node: A node instance representing the execution of the start node is placed on the execution queue. From here

---

<sup>2</sup> Service objects may be empty, not doing anything.

- on, the navigation takes place whenever a step finishes its execution and returns control to its executor.
2. When a step finishes, the outgoing control flows from its node are iterated. For each control flow evaluate its transition condition. If it is false, ignore the control flow. If it is true, traverse to the target node and check if the node can be scheduled for execution. If the target node has a single incoming control flow, then it can be scheduled; otherwise:
    - (a) If the target node is an AND-node then all its incoming control flows must have been traversed already in order to proceed (i.e., the last transition to the node has just been made).
    - (b) If the target node is an OR-node then execution will not proceed, unless this is the first transition to the node—this causes all other transitions to the target node to have no effect and allows execution to proceed immediately.
  3. When the next node is ready to be executed, a node instance is placed on the execution queue.
  4. The workflow execution ends when a node finishes its execution, there are no control flows to apply, and there are no other nodes still in execution.

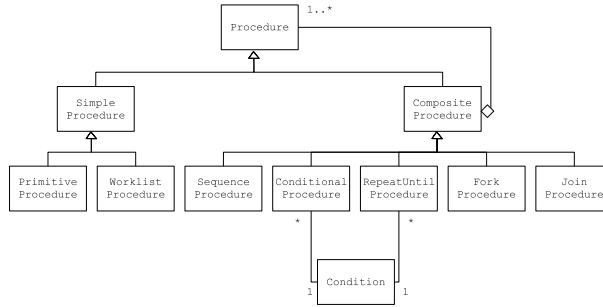
### 4.3 Frontend Definition

The choice of a frontend language is like the choice of a high-level, structured programming language, whereas the choice of a backend language is more like the choice of a low-level, unstructured byte-code language. At the frontend we rather have some modelling rules enforced, to get a more understandable and structured model. The backend language described in the Sect. 4.1 is a graph-based representation of a workflow, which therefore allows for the creation of unstructured workflow models (e.g., jumps in and out of loop's bodies, splitting with inconsistent merging, etc). This freedom and expressiveness is required to support the different frontend languages available. The frontend language we implemented is a subset of the typical workflow definition languages, that supports only commonly used structured workflow patterns [10].

The frontend language we describe here is based on the work by Manolescu, in his PhD Thesis [7]. The key abstraction of this particular frontend language is the *Procedure*. A tree of hierarchically composed procedures defines a workflow. Manolescu proposes control structures for procedures which are similar to those available in structured programming languages.

Procedures are built using the Composite [4] design pattern to abstract the concrete type of the procedures that constitute the workflow definition, thus allowing the easy extension of the framework with new procedures. The diagram in Fig. 1 shows the complete hierarchy. We will discuss some of them. There are two main sub-types of procedure: *Simple Procedures* and *Composite Procedures*.

Simple procedures are leaf nodes in the tree of procedures that constitute the workflow definition. These procedures are associated with domain-specific tasks performed outside the framework, i.e., they represent workflow steps which



**Fig. 1.** Meta-model of the frontend definition

must be delegated for execution by the domain workers (either applications or humans).

Composite procedures are used to represent the structures that manage the flow of control (such as sequencing, and branching). We defined five types of composite procedures: (1) *Sequence Procedures*, which represent the execution of all their children procedures in the sequential order in which they are stored; (2) *Conditional Procedures*, which enable the control flow to change by having a condition that specifies one of two mutually exclusive paths to take; (3) *Fork Procedures*, which spawn the parallel execution of all their children and wait for the first to finish; (4) *Join Procedures*, which, like the previous one, spawn the parallel execution of all their children, but wait for all of them to complete execution before completing; and (5) *RepeatUntil Procedures*, which hold one procedure as their body and cyclically execute it until a given condition is achieved.

#### 4.4 Mapping the Frontend to the Backend

In this section we describe some of the transformation rules used to generate the backend definition from the frontend based on procedures. The compilation starts in the root of the tree of procedures and is defined recursively: Each composite procedure compiles its children (which generates part of the backend graph), and then link some of the resulting nodes, depending on the procedure semantics. The compilation of a leaf procedure produces only one node, with the appropriate service object.

All composite procedures are represented by, at least, two graph nodes containing between them the compiled graphs that correspond to the children procedures. By default, transition conditions always evaluate to `true`, meaning that the transition should always be made to the target node (the exceptions are the Conditional and RepeatUntil procedures).

The entire compilation process is quite straightforward, but its details are outside the scope of this paper. To illustrate the compilation method, Fig. 2 shows some examples of workflows represented in procedure-like fashion and its corresponding compilation to the backend language.

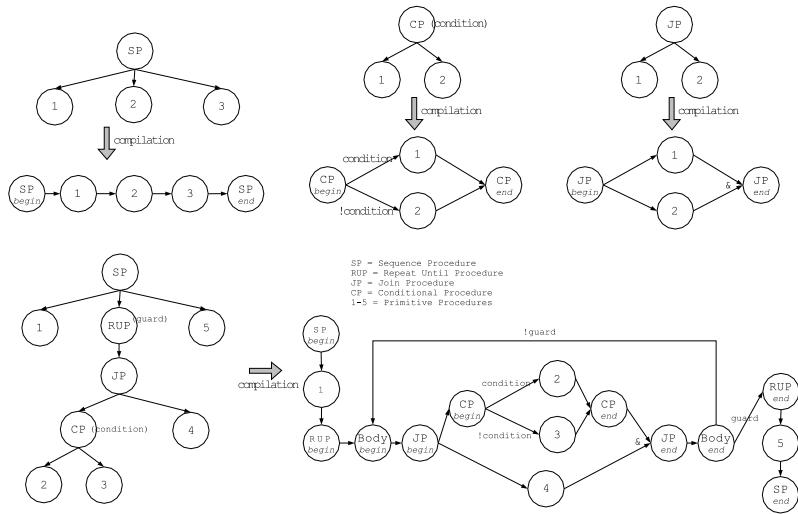


Fig. 2. Frontend representations and the respective compilations to the backend.

## 5 Discussion

Some previous work approached, already, the construction of object-oriented WfSs. For instance, [7] proposes a new workflow architecture for software developers that focuses on customising the workflow features and composing them; and in [8] a lightweight workflow kernel is proposed, with some emphasis on distributed execution capabilities. However, none of this work approaches the architectural issue of supporting different languages for representing workflows. Therefore, all of them are limited on the workflow languages they support. To the extent of our knowledge, there is no other WfS that follows our approach.

On the other hand, our workflow framework implementation raises some interesting questions. For instance, is the backend language we designed expressive enough? We took a pragmatic approach in our development: We chose to incrementally and iteratively develop the language. We began by identifying the core/common workflow concerns expressed in most languages and defined a language to support that. Later, if the need arises, the backend language can be extended or reviewed.

Recently, some authors performed a very complete study on the expressiveness of workflow languages and its application to typical workflow patterns [6, 10]. One result of this study was the proposal of a workflow language [1] with a representation based on Petri nets. Despite its high expressiveness, we do not consider it adequate as a frontend language. For many users it is far more complex than what they need. However, it could be used as the backend language in our approach. A drawback is that this language is very complex, which may difficult its implementation in a WfS.

## 6 Conclusion

We presented a technique for dealing with the variations in workflow languages, which consists in separating the language used to model workflows from the language used by the workflow system to execute them. With this separation, the core system behaves like a workflow virtual machine. This technique was applied to the development of an object-oriented workflow framework that provides workflow concepts to software developers. We consider that this technique can be applied to the development of any workflow system to increase its reusability when dealing with changes in the workflow languages. It also presents a solution for developers that intend to provide support for more than one language.

It would be interesting to exercise the compilation of more frontend languages to our backend language, as a way to test the expressiveness of the backend. Currently, we are also undergoing work for creating modular functional extensions to our framework (e.g. monitoring, access control, dynamic changes of the workflows). These modules will allow us to assess the savings we can obtain by developing code for the backend that will automatically provide the same functionality to many frontend languages.

## References

1. W.M.P. van der Aalst and A.H.M. ter Hofstede. Yawl: Yet another workflow language. Technical Report FIT-TR-2002-06, Queensland University of Technology, 2002.
2. Assaf Arkin. Business process modelling language, November 2002.
3. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services.
4. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
5. David Hollingsworth. *The Workflow Reference Model*. Workflow Management Coalition, Jan 1995. Document Number TC-00-1003.
6. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2002.
7. Dragos-Anton Manolescu. *Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.
8. P. Muth, J. Weissenfels, M. Gillmann, and G. Weikum. Mentor-lite: Integrating light-weight workflow management systems within existing business environments (extended abstract), 1998.
9. W. van der Aalst. Don't go with the flow: Web services composition standards exposed, 2003.
10. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. QUT Technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane, 2002. (Also see <http://www.tm.tue.nl/it/research/patterns>).