UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

A Workflow Virtual Machine

Sérgio Miguel Martinho Fernandes
(Licenciado)

Dissertação para obtenção do grau de Mestre
em Engenharia Informática e de Computadores

Orientador:	Doutor António Manuel Ferreira Rito da Silva
Presidente:	Doutor Pedro Manuel Moreira Vaz Antunes de Sousa
Vogais:	Doutor João José da Cunha e Silva Pinto Ferreira
	Doutor António Manuel Ferreira Rito da Silva

Resumo

O suporte fornecido pelos sistemas de gestão de fluxos de trabalho (*workflow*) tem vindo a aumentar ao longo dos anos. Estes sistemas evoluíram desde o suporte à automatização de tarefas individuais, até à coordenação de processos-chave das organizações. A investigação na área da tecnologia de *workflow* tem sido realizada tanto na comunidade académica como empresarial. Todo este interesse conduziu à proliferação de propostas de normas e à implementação de muitos sistemas heterogéneos.

Uma das consequências directas é a disparidade de linguagens de modelação de *workflow* e a dificuldade, por parte dos programadores destes sistemas em escolher uma linguagem, suportá-la ao longo das suas muitas evoluções e conter os custos de manutenção causados pelas frequentes mudanças.

Nesta tese propõe-se um modelo para o núcleo de um sistema de gestão de *workflow* que: 1) Visa minimizar o impacto causado no sistema pelas alterações introduzidas na linguagem; 2) Procura ser facilmente extensível com novas funcionalidades de uma forma independente da linguagem suportada.

O modelo proposto separa a linguagem utilizada para modelar (*front-end*) da linguagem utilizada para executar (*back-end*) *workflows*. São descritos os módulos de definição e de execução da máquina virtual de *workflow* que suporta a linguagem de *back-end*. Em seguida descreve-se um sistema que foi implementado tendo por base o modelo proposto e termina-se efectuando uma avaliação das qualidades expressivas da linguagem de *back-end*, baseada nos padrões de *workflow*.

Abstract

The support provided by workflow management systems has been increasing over the years. These systems have evolved from automating individual tasks, to coordinating key organisational processes. Research in workflow technology has been conducted both in the academic community and within business organisations. All the interest has led to the proliferation of standards proposals and to the implementation of many heterogeneous systems.

One of the direct consequences is the disparity of workflow modelling languages and the difficulty that workflow systems' developers have in choosing one language, supporting it over its many evolutions, and restraining the maintenance costs caused by the frequent changes.

In this thesis we propose a model for the core of a workflow management system that: 1) Seeks to minimise the impact caused on the system by changes to the modelling language; 2) Aims to be easily extensible with new functionalities in a language-independent fashion.

The proposed model separates the language used to model (front-end) from the language used to execute (back-end) workflows. We describe the definition and the execution modules of the workflow virtual machine that supports the back-end language. We then describe a system that was implemented based on the proposed model and we finish by performing an evaluation of the expressive qualities of the back-end language, based on the workflow patterns.

Palavras-chave

Sistemas de Gestão de Fluxos de Trabalho

Linguagens de Definição de Fluxos de Trabalho

Máquina Virtual de Fluxos de Trabalho

Padrões de Fluxos de Trabalho

Separação de Facetas

WorkSCo

Keywords:

Workflow Management Systems

Workflow Definition Languages

Workflow Virtual Machine

Workflow Patterns

Separation of Concerns

WorkSCo

Acknowledgements

I would like to thank my adviser, Professor António Rito da Silva, for all his guidance over these years. I feel that I have learned much from him. Also, more than an adviser, he has also become a friend.

I thank everybody who was involved in the WorkSCo project, especially to the following: Paulo Dias with whom I developed the initial prototype; Jorge Martins, Pedro Vieira and Dulce Domingos who have extended the system with new and relevant features; Pedro Santos who experimented with WorkSCo in his graduation project; and João Cachopo with whom I wrote an article on this thesis' subject. They have all taken their time to listen to my ideas and to present me with new ones. All the discussions we had, proved to be quite enlightening.

I also thank to all the members of INESC-ID's Software Engineering Group. It has been my pleasure to work with all of them. Thank you all for making me feel that this is the right group to be part of.

Thank you to the members of INESC-ID's Spoken Language Systems Lab and Distributed Systems Group. You are not only great colleagues but also excellent people and you have all contributed in some way to this work.

Many thanks to all my good friends, who kept believing I would finish this thesis. Even when I though I was loosing hope I found it in their encouraging words. Especially to Ricardo Português for putting me up to this, and to Rui Nunes for keeping me at it up until the end, I thank you.

And last, but certainly not least, I thank my mother and grandmother, for their unconditional support, not only throughout this thesis, but for my entire life.

Lisbon, September 2005

Sérgio Miguel Martinho Fernandes

Contents

List of Figures

1 Introduction

Since the 1970's, when the first *Office Information Systems* started to be developed, interest in workflow technology has been constantly increasing. The initial aim was to increase the productivity of individual workers, by structuring their daily routines. Afterwards, focus shifted from automating individual work, to coordinating business processes of one organisation as a whole. The evolution of computer systems has contributed much to this. Organisations became interested in the systems' capacities to handle electronic information. More recently, organisations have started to look for support for inter-organisational processes.

Even though a scenario of full interoperability between organisations seems appealing, there have been some difficulties. One of the difficulties stems from the fact that there is a proliferation of heterogeneous workflow systems. Developers have been solving interoperability issues almost in a case-by-case fashion. Some groups have proposed standards for workflow, but they haven't been able to suit everyone's needs.

1.1 Problem Definition

One of the challenges that the development of a workflow system faces is the choice of the modelling language, in which workflows will be represented. Possibilities range from supporting one of the many available standards, to implementing a proprietary modelling language. Each of the choices carries consequences: The standards for workflow modelling are not yet stable and there are many proposals to choose from, which is almost like having no standard at all. Choosing a proprietary language hinders interoperability and the reuse of other workflow models.

Additionally, there are conflicting forces on the requirements for such language. On the one side, the modellers want a user-friendly, feature-rich, and expressive language. On the other side the developers are concerned with the system's maintenance and how will it stand any future changes to the language: Therefore, they rather have a smaller and simpler language, without redundancy. The resulting system is usually a compromise between these forces.

1.2 Our Approach

We believe that different domains have different modelling requirements and therefore a universal modelling language that suites all needs is either not feasible or yet to be created. This might account for the proliferation of available systems, each with its own language. Instead of trying to create a standard language, we choose to accept the diversity of languages.

Our goal is, therefore, to develop a workflow system that can be easily adapted to support changes in the modelling language or even to support, in the same system, the execution of models each of them written in a different language. Changing the workflow language should have the least impact on the system. This is the main focus of this dissertation.

1.3 Contributions

The main contributions of the work presented in this thesis are the following:

- **Proposal of a model for a Workflow Virtual Machine:** We propose to split the core of the workflow system in two layers. This design decouples the language used to model workflows (front-end language) from the language used to execute them (back-end language). The lower layer is a reusable *Workflow Virtual Machine*, which supports its own definition and execution of workflows.
- **Description of the back-end modelling language**: We provide a complete description (both of the meta-model and of its execution semantics) for the language operated by the *Workflow Virtual Machine*.
- **Description of one front-end modelling language**: We define one possible structured front-end language with control primitives that are similar to those available in programming languages. We also provide the rules from transforming it into the back-end language.
- **Implementation of the core of a Workflow Management System**: We implemented the core of an open source *Workflow Management System* [Wor], based on the concept of the *Workflow Virtual Machine*, and using the back-end and front-end languages mentioned previously. This system has been applied, at least, in two European research projects [COM, ACE].
- **Support for the realisation of other theses**. Other researchers have investigated and extended the core implementation of the *Workflow Management System* with functionality in the areas of Adaptive Workflow Management, Access Control, and Monitoring.
- **Publication in international conferences**. As a direct result of the work on this thesis, we have published an article [FCS04] about supporting evolution of workflow definition languages, which is based on the concept of the *Workflow Virtual Machine*.

1.4 Dissertation Structure

Chapter 1—Introduction. In this chapter, we present the context for this thesis, state the problem it addresses, and describe the main contributions of the work.

Chapter 2—Workflow Management Systems. This chapter presents the related work that constitutes the basis for this thesis. It is an overview of the most relevant concepts regarding workflow management systems, focusing especially on the Workflow Reference Model.

Chapter 3—Evaluation of Workflow Management Systems. We start this chapter by proposing a framework for evaluating workflow systems based on the concepts supported by their meta-models. We divide the framework in five categories with a total of twenty one characteristics. We then evaluate six systems using the proposed framework.

Chapter 4—The Workflow Virtual Machine. This chapter describes the proposed model for the *Workflow Virtual Machine*, which separates the language used to model workflows from the language used to execute them. We focus on the meta-model for the back-end definition and on the algorithm applied by the execution module.

Chapter 5—Implementation. This chapter describes the implementation of WorkSCo, which is a *Workflow Management System* built upon the concept of the *Workflow Virtual Machine* from the previous chapter. We provide the description of a concrete front-end language, as well as the mechanism for compiling it to the back-end language. We end by addressing the execution-related aspects.

Chapter 6—Evaluation. This chapter studies the suitability of the back-end language. We evaluate its expressiveness by analysing typical workflow patterns. For each workflow pattern, we provide a brief description of the pattern and a discussion of how it can be modelled in the back-end language.

Chapter 7—Concluding Remarks. This chapter ends the dissertation giving an overview of the work presented. It summarises its main results, and indicates possible directions for future work.

2 Workflow Management Systems

It is generally accepted that the *Office Information System* (OIS) was the predecessor of today's workflow system. The first OISs were developed in the late 1970's [SZ01]. The pioneers included the SCOOP project [Zis77], which was a system oriented towards the automation of office procedures, and OfficeTalk [Ell79], which was designed to investigate alternatives for distributed control and data manipulation within a communal computing environment. Up until the mid-1980's the research interest in this field continued, but few applications succeeded. For the rest of the decade, the initial optimism withered, as the OISs' promise "to reduce the complexity of the user's interface, control the flow of information, and enhance the overall efficiency of the office" wasn't fulfilled [EN80, Nut96].

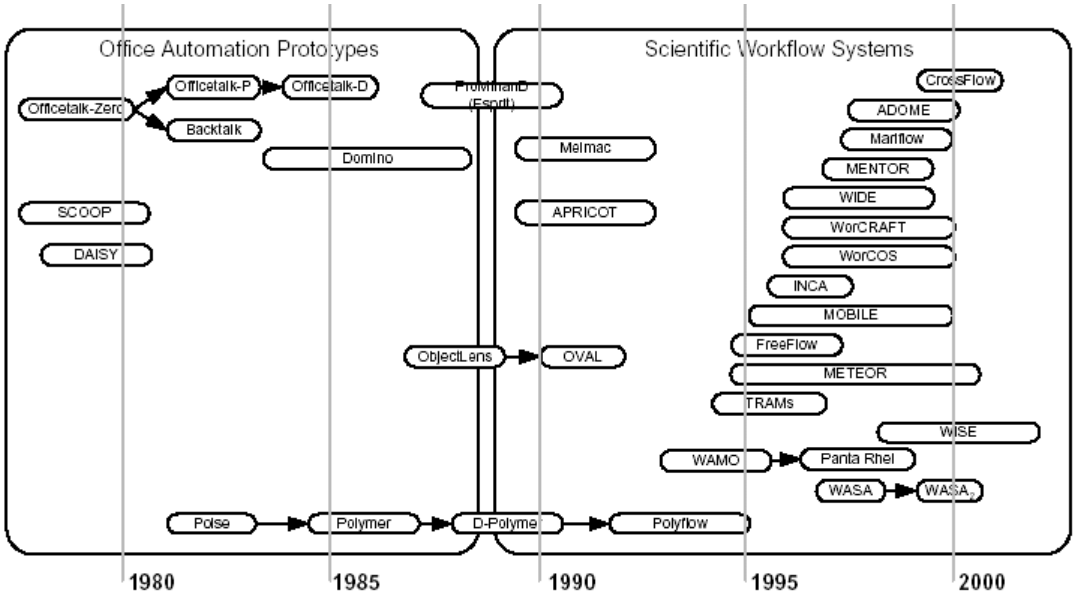

Figure 2.1: History of workflow research (from [MÖ4]).

It wasn't until the 1990's that the research interest in process automation resurged, as Figure 2.1 demonstrates. Whereas OISs put more emphasis on automating individual activities, in the 1990's the focus shifted to capturing and supporting business processes associated with critical organisational functions that connected several individuals [AS94].

With the rapid growth of the systems available, both from research and commercial areas, the need to define a standard that facilitated the interoperability became clear. It was in this context that the *Workflow Management Coalition* (WfMC) [wfm] was created. It is composed by companies, research institutes, and users.

Even though the WfMC defined a glossary of workflow concepts [WMC99a] and created the *Workflow Reference Model* (WfRM) [Hol95], they have been unable to impose a standard. Today, there

are numerous competing standards available (e.g. XPDL [Tha01], BPEL4WS [ACD+03], BPML [Ark02], WSFL [Ley01], XLANG [Tha01], and WSCI [AAF+02]) and the so desired interoperability is yet to be achieved [Aal03].

2.1 Basic Concepts

A *workflow* represents the operational aspects of a work procedure: The structure of tasks and the applications and humans that perform them; the order of task invocation; task synchronisation and the information flow to support the tasks; and the tracking and reporting mechanisms that measure and control the tasks. Workflow automates the process logic. Humans and software applications (workflow resources) perform workflow tasks, thus implementing the task logic. This separation of process and task logic intends to allow workflow users to modify one without affecting the other. It also promotes software reuse and the integration of heterogeneous software applications. Therefore, workflow automates processes that fit a 2-tier model, as Fig. 2.2 illustrates. In this model, the flow tier contains the process logic, whereas the work tier corresponds to the basic process tasks. The flow tier controls and automates the coordination of the work tier. Workflow management provides an environment for the definition and enactment of the process logic.

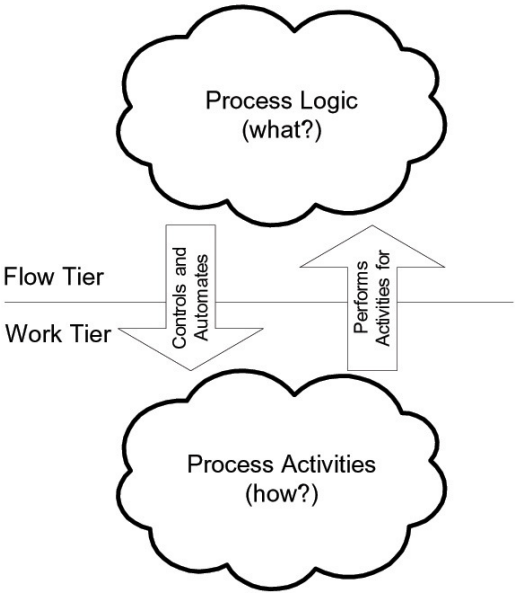

Figure 2.2: The process logic and activities are partitioned on the flow and work tiers.

The workflow area is concerned with the automation of procedures where documents, information or tasks are passed between participants according to a defined set of rules to achieve, or contribute to, an overall business goal. According to the WfRM a *workflow* is:

> The computerised facilitation or automation of a business process, in whole or part.

The workflow literature refers to the software that enables people to define and execute workflows as a *Workflow Management System* (WfMS). A WfMS automates processes by managing tasks

and resources. The WfRM provides the following definition:

> A *Workflow Management System* completely defines, manages and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic.

All WfMSs may be characterised as providing support in three functional areas:

- The *build-time functions*, concerned with defining, and possibly modelling, the workflow processes and its constituent activities;
- The *runtime control functions* concerned with managing the workflow processes in an operational environment and sequencing the various activities to be handled as part of each process;
- The *runtime interactions* with human users and IT application tools for processing the various activity steps.

Fig. 2.3 illustrates the basic characteristics of a WfMS and the relationships between these main functions.

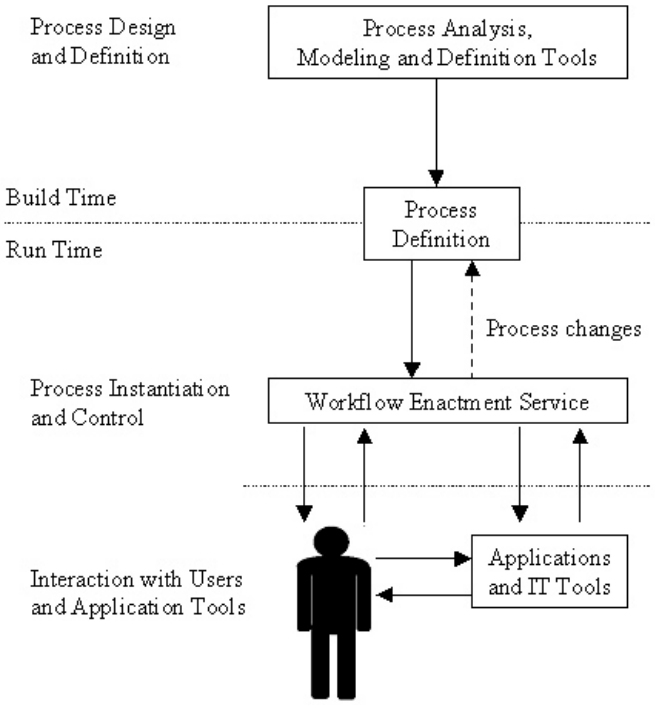

Figure 2.3: Workflow Management System's characteristics.

2.1.1 Build-time Functions

The build-time functions are those whose result is a computerised definition of a process. During this phase, a process is translated from the real world into a formal, computable definition by the use of

one or more analysis, modelling and system definition techniques. The resulting definition is called a process model or a workflow model.[1]

A process definition normally comprises a number of discrete activity steps, with associated computer and/or human operations and rules governing the progression of the process through the various activity steps. The process definition may be expressed in textual or graphical form or in a formal language notation. Some workflow systems may allow dynamic modifications to process definitions from the runtime operational environment, as indicated by the feedback (dashed) arrow in Fig 2.3.

The members of the WfMC do not consider the support for the initial creation of process definitions to be an area of standardisation. Rather, this is considered a major distinguishing area between products in the marketplace. However, the result of the build-time operation, i.e., the process definition, is identified as one of the potential areas of standardisation to enable the interchange of process definition data between different build-time tools and runtime products.

2.1.2 Runtime Control Functions

At runtime, the process definition is interpreted by software that is responsible for creating and controlling operational instances of the process, scheduling the various activities within the process and invoking the appropriate human and IT application resources, etc. These runtime process control functions act as the linkage between the process, as modelled within the process definition, and the process as it is seen in the real world, reflected in the runtime interactions of users and IT application tools. The core component is the basic workflow management control software (or "engine"), responsible for process creation and deletion, control of the activity scheduling within an operational process and interaction with application tools or human resources. This software might be distributed across several computer platforms, e.g. to cope with processes that operate over a wide geographic basis.

2.1.3 Runtime Interactions

Individual activities within a workflow process are typically concerned with human operations, often realized in conjunction with the use of a particular IT tool (for example, form filling), or with information processing operations requiring a particular application program to operate on some defined information (for example, updating an orders database with a new record). Interaction with the process control software is necessary to execute operations such as to transfer control between activities, to determine the operational status of processes, to invoke application tools, and to pass the appropriate data. There are several benefits in having a standardised framework for supporting this type of interaction, including the use of a consistent interface to multiple workflow systems and the ability to develop common application tools to work with different workflow products.

[1]The difference between a process model and a workflow model is not relevant for this thesis. However, some authors distinguish a process from a workflow. They consider that a process might yet not be executable by a WfMS, because it lacks some details, which are already present in a workflow. The WfMC refers to the executable definition as a process.

2.2 The Workflow Reference Model

The WfRM [Hol95] is an effort from the WfMC to provide a description of a standard architecture for a WfMS. The reference model identifies common concepts involved in the construction of a WfMS, defines several components and provides a common terminology.

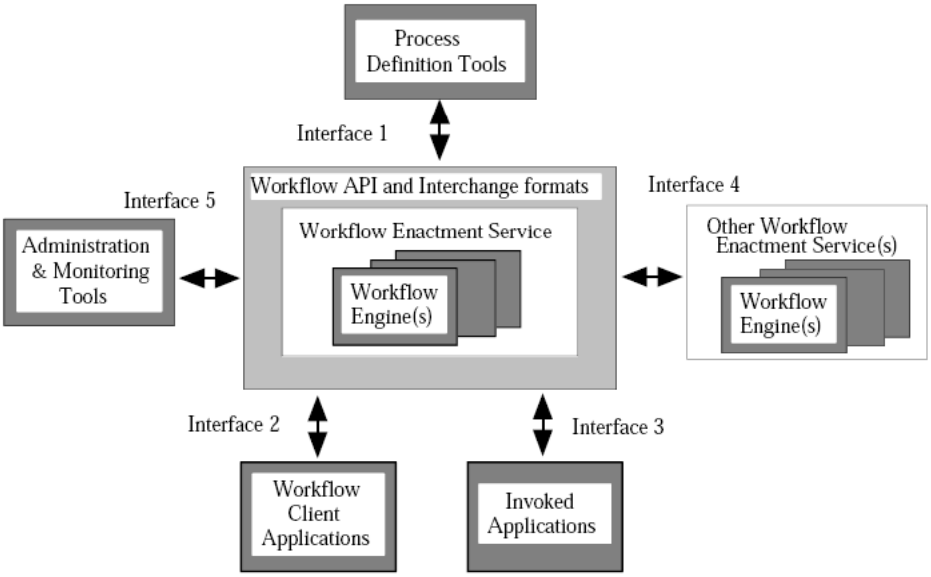

Figure 2.4: Workflow Management System—Components and Interfaces.

The WfRM divides a WfMS into 6 major components, which are shown in Figure 2.4.

The *Workflow Enactment Service* (WfES) provides the runtime environment in which process instantiation and activation occurs. The core element of this component is the workflow engine, which is responsible for interpreting and activating the process definition and interacting with the external resources necessary to process the various activities. The workflow engine is, in its simplest form, a state machine where individual process or activity instances change states in response to events.

The *Process Definition* component supports the creation of process definitions and communicates them to the WfES. A process definition contains all the necessary information about the process to enable its execution by the WfES. This includes information about its starting and completion conditions, constituent activities and rules for navigating between them, user tasks to be undertaken, references to applications which may have to be invoked, and definition of any workflow relevant data which may need to be referenced.

The *Invoked Applications* component handles the execution of activities whose enactment is provided by a software application. This component's responsibilities are to provide an interface for the workflow engine that abstracts the details of the invocation. It is usually the case that the invoked application receives and/or produces data. This component is also responsible for the data transmission between the workflow engine and the application.

The *Workflow Client* is a component dedicated to the runtime interactions with workflow participants in those activities which require human resources. Whenever user interactions are necessary

within the process execution, the workflow engine places a work item on a worklist. Worklist handlers control the interactions between worklists and users. Worklists are containers for work items, which represent units of work that users need to perform as part of the process enactment. The work executed by the user may be supported by some software tool; in this case the Workflow Client component might interact with the Invoked Applications component by typically invoking some application which provides a user interface.

The *Workflow Interoperability* component deals with the communication and the synchronisation between distributed workflow engines that execute interconnecting parts of a process. This component addresses issues such as connected vs. disconnected operation modes, centralised vs. distributed process definition, and synchronous vs. asynchronous (i.e. nested vs. parallel) activity execution. In regard to this component, the WfRM does not commit itself to a proposal for a standard. Instead, it presents a variety of interoperability scenarios and discusses the requirements imposed on the intervening systems.

Finally, the *Administration* component intends to standardise administration and monitoring functions, thus allowing one system's management application to work with another's engine(s). Typical functions include user, role, and audit management, resource control, and process supervisory tasks.

Additionally to the identification of the components, the WfMC has been detailing each of the Interfaces 1 to 5 [WMC02, WMC98b, WMC99b, WMC98a].

2.3 Workflow Modelling

A *Workflow Definition Language* (WfDL)[2] supplies the constructs defined in the underlying meta-model, enabling the definition of workflows. Figure 2.5 shows the concepts identified by the WfMC as being the basic elements used in a workflow definition.

The *Workflow Type Definition* represents the whole process being modelled. It has a name that identifies the model, a version number for differentiating between two models of the same process, the start and termination conditions for the process, and security and audit data.

A workflow type definition consists of a set of *Activity* descriptions. The concept of activity is central to the workflow meta-model. An activity represents a piece of work to be performed. It has an identifier (usually a unique name), and a type that indicates the characteristics of the activity (e.g., atomic/composite, manual/automatic). Activities may have pre- and post- execution conditions that state under which circumstances the activity can be successfully executed. Activities usually consume and produce data relevant for the workflow's execution (*Workflow Relevant Data*). Depending on their type, activities can be executed manually or automatically. In either case, they may be associated with the invocation of some application (*Invoked Application*). The invoked application typically has access to the same data as the activity.

Some activities are restricted to being executed by certain types of resources in the organisation. This is expressed with an association between the activity and a *Role*. In this context a role is the representation of some organisational function or entity.

[2] Also known as Workflow Modelling Language.

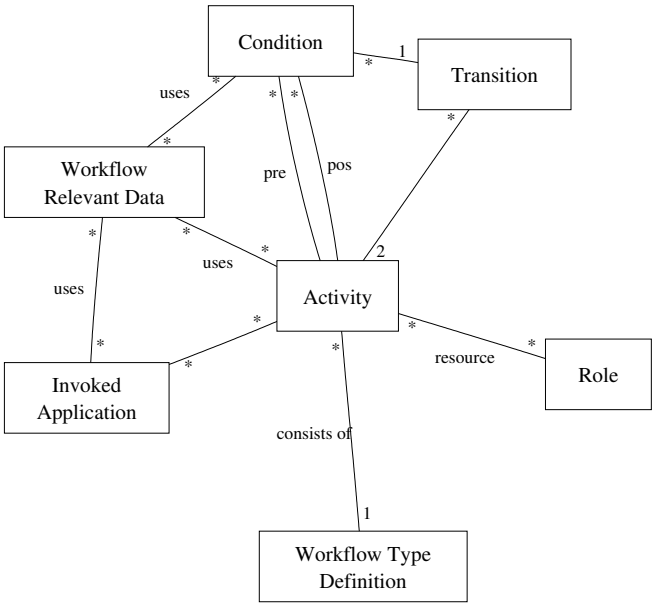

Figure 2.5: Basic Workflow Definition Meta-model.

A *Transition* models a restriction to the order of execution of activities. It imposes a strict order between activities and it also defines the *Condition*s under which the transition to the target activity may occur. It is common for the condition to use workflow data.

2.3.1 Workflow Perspectives

There are several aspects or concerns to a workflow meta-model. The literature [CKO92] identifies these aspects and categorises them into workflow perspectives.

Functional Perspective. The functional perspective defines what has to be done. It contains the definition of the workflow's activities. The activities can be atomic or composite. Composite activities include other activities and define a hierarchy in the workflow definition. Usually domain-specific work is performed in the atomic activities.

Behavioural Perspective. This perspective describes the order of execution of the activities, thus controlling the flow. It is either supported by composite activities or specific control flow constructs, such as transitions.

Informational Perspective. The informational perspective comprises the data and the data flow. The data is consumed and produced by the workflow activities. The data flow describes which data is passed along the workflow from/to which activity. It is common to divide data in two types: *Control data* and *Application data*. The former refers to data handled internally by the workflow engine to control the execution mechanism. The latter corresponds to the data effectively manipulated by the domain entities during the execution of the activities and may also be used to model decisions that affect the execution.

Organisational Perspective. This perspective models who performs the work in the workflow. It supports an (often simplified) organisational model that identifies resources and models their assignment to the activities. The assignment can be performed indirectly through roles, i.e., the model assigns roles to activities instead of directly assigning resources to activities. At runtime the resources that have the required role may perform the activity.

2.4 Workflow Taxonomy

There is more than one proposal for classifying workflows. In this section, we summarise three different classifications proposed in the literature.

The first, and perhaps the most divulged considers four workflow types: *ad-hoc*, *production*, *administrative*, and *collaborative* workflows. The initial proposal mentioned only the first three categories [McC92]. The fourth was added later [Moh96]. Ad-hoc workflows describe exceptional or unique situations. They are generally defined to be executed once or a very limited number of times. These workflows are simple and have little structure. The coordination of the activities requires human intervention, i.e. the control flow model is not completely defined. Production workflows describe processes which are critical to the organisations. They are well defined, much structured and their execution usually involves many activities that access several external systems. Administrative workflows describe bureaucratic processes, which have a well-know description. These workflows are usually repetitive and most of the times their execution can be automated. Collaborative workflows are defined mainly by the great number of participants involved in the process and the many interactions that exist between them. They are very dynamic workflows and sometimes they are defined as they are being executed.

Another classification scheme distinguishes between *human-oriented*, *system-oriented* and *transactional* workflows [GHS95]. In human-oriented workflows people execute and coordinate their activities. They are responsible for the coherence of the workflow. The WfMS supports the coordination and collaboration between people. System-oriented workflows are highly automated. They are performed by software systems that execute intensive operations and specialised software tasks. The process can usually perform error recovery to maintain reliability and coherence. Transactional workflows are performed both by humans and external software systems. The emphasis is on the fact that these workflows provide transactional properties that derive from database systems, such as atomicity and isolation at individual activity or intra and inter-workflow levels.

Finally, [Kie02] classifies the workflows based on the concepts supported, the evaluation strategy used, and the syntactic restrictions imposed. Four categories are considered: *standard*, *safe*, *structured*, and *synchronising* workflow models. The standard workflow models correspond to supporting the basic control flow patterns and multiple instances. No restrictions are imposed on loops and termination. States are not supported. The safe workflow models exclude the idea of multiple instances. In these workflows it is not possible that at some point in time there exit multiple instances of the same activity in the same workflow instance. Structured workflow models form a subclass of standard workflow models where every AND/OR-split has exactly one corresponding AND/OR-join and ar-

bitrary loops are not allowed (i.e. loops should have one entry and one exit point). Synchronising workflow models can be thought of as propagating true and false tokens. If an activity receives a true token, it will execute. If it receives a false token, it will simply pass it on. Branches of choices not chosen propagate false tokens, while branches that are chosen propagate true tokens. Synchronisation points await tokens from all incoming branches and, depending on their type, either 1) propagate a true token if it has received only true tokens, and a false token otherwise, or 2) propagate a true token if it has received at least one true token, and a false token otherwise.

2.5 Workflow Features

Although there are several features associated with workflow systems, these systems share a small set of common features, which follow:

Flow-Independence: due to the participation of different concerns in an application, developers would like to write software such that every design decision is encapsulated into a component, thus separating concerns. Ideally, this would allow changes in one component not to affect other parts of the application. Workflow enables developers to separate the flow between the application's components from the application (i.e., the process). This is so, because the process logic is separated from the tasks. As already referred, this allows the separate modification and evolution of process logic and process tasks.

Domain-Independence: the described 2-tier model keeps the workflow execution independent from the application domain. This characteristic makes workflow technology applicable to a large number of application domains. Thus, applying workflow to a particular application domain requires providing components that perform domain-specific work. However, the execution environment provided by the infrastructure supporting the process logic can be completely reused.

Monitoring and History: the separation of process logic from the application components enables workflow to tap into the process level and collect information about its execution, which can be displayed at run time and after the process is completed. Workflow monitoring and history are typical of workflow systems. Flow-independent applications that implement their process models with workflow technology can use these features at no extra cost.

Manual Intervention: this feature enables workflow systems to handle exceptions and unique situations by allowing users to override the process definition and manually change the course of the process. Providing various degrees of flexibility is one of today's current workflow systems aims.

3 Evaluation of Workflow Management Systems

WfMSs started to appear in the 1980's. Today there are many WfMSs available and the number of commercial and open source initiatives keeps growing [Bae04, Top, ope]. However, despite the efforts of the WfMC to provide a standard, workflow products are mostly not interoperable. We identify at least two reasons that contribute to such fact: On the one hand, it is common among WfMSs, to find discrepancies in the meta-models and in the APIs. It is easy to find different products that give different names to the same concept (e.g. some call "Activity" to what others call "Step") or provide unique concepts that don't exist in other languages (e.g. anticipated execution of activities); on the other hand, there are many groups developing competing standards and none has yet emerged as the winner (if one ever will) [Aal03], thus scattering the developers of WfMS, which have to opt either for a dubious winner or for developing their own specific meta-models and interfaces. Yet, many of the concepts described by the WfRM are present in most implementations.

From a modelling point of view, our interest is on the kind of workflows that can be represented and executed in a given WfMS, i.e., we want to assess some of the expressiveness of the workflow meta-model through the analysis of the workflow concepts supported by the modelling language of the WfMS.[1]

In this chapter we perform an evaluation of some WfMSs, based on their meta-models. There were many systems to choose from. We chose the presented systems (jBPM, OSWorkflow, Twister, YAWL, WfMOpen, and DWFMS) because they have been under active development, they are used in commercial applications, they are well documented, and they are open source. They also cover a large spectrum of approaches to WfMS development. There are many other WfMSs available. The following list gives a general idea: Bonita, Bossa, Breeze, Enhydra Shark, OBE, OFBiz, OpenFlow, PowerFolder, Taverna, Werkflow, WorkFlow ToolKit, and more.

3.1 Key Dimensions of the Framework

The key dimensions that we propose for evaluating WfMSs derive both from the WfRM and from evaluations performed by other authors. In [M99] the author analysis two WfMSs. However, he only considers the organisational part of the meta-model on his evaluation. Similarly, [BKKR03] deals with the inter-organisational aspects of the meta-models, focusing only on Interface 4 of the WfRM. In [LS97] a more broad approach is taken. The analysis does not favour any specific part of the meta-model; instead, each of the WfMSs is analysed according to a set of very well defined dimensions. Yet,

[1] Another possible approach to compare WfMSs would be to do an evaluation based on functional characteristics [DLTW97]. This would no doubt be interesting, but perhaps more relevant if we were interested in installing one such system to satisfy our business requirements.

the evaluations mentioned previously, refer mostly to WfMSs or languages that are outdated. Some of them are no longer under development. It should also be noted that [AHKB03] contains a very good meta-model analysis for some WfMSs. The analysis is based on a solid study of the existing workflow patterns, but it only focuses on the control flow aspects, leaving out other perspectives. Following, we present our proposed key dimensions that make up the framework for comparing the WfMSs with respect to their meta-models. We group the dimensions into five categories according to the aspects that they refer to.

3.1.1 Meta-model

This category deals with the meta-model's general characteristics. They affect the overall structure of the workflow models.

Language. A relevant issue regarding the meta-model, is whether the WfMS uses a proprietary language or one of the proposed standards.

Structure. The same workflow model can be represented in different data structures. However, when a language is specified, its creators tend to have a data structure in mind, in which it is more straightforward to represent the workflow models. Some of the most used structures are graphs, trees, and Petri nets.

Basic elements. The basic elements of a workflow meta-model are the core concepts around which a workflow model is created, such as activity, state, and step.

3.1.2 Functional Perspective

Every workflow model contains at least two important pieces of information: The *work* and its *control flow*. The work refers to the concrete tasks represented in the model, which have to be executed either by software applications or by humans. The control flow is what imposes the order in which work needs to be executed. In the functional perspective category we evaluate the aspects related to how work can be modelled.

Manual work. This characterises the meta-model's support for representing work that involves human workers. Not all meta-models support this, because some of them are only geared towards orchestrating fully automatic business processes or assume that manual work must be handled outside the model.

Automatic work. Given that WfMS are software products, it is more than natural that they support invocation of other software systems which automatically perform some work. This is, therefore, the most common feature.

Domain integration. The WfMS often needs to invoke external systems or software components to accomplish automated or manual tasks. These invocations need to be specified in the model. Most meta-models only support the description of invocations for a certain type of technology, e.g. Java.

3.1.3 Behavioural Perspective

All characteristics evaluated under this category deal with the ability of the meta-model to represent coordination of the work and how to react given its possible outcome. Do not confuse the characteristics considered in this category with an analysis based on workflow patterns: We are assessing which control flow elements does the meta-model have, whereas the study based on workflow patterns checks which control flow semantics can be achieved with the control flow elements provided by the meta-model.

Sequence. A sequence is an ordered list of tasks. Each task should be executed only after the previous task completes. The first task on the list can be immediately executed, when the sequence begins. Some meta-models fail the execution of the sequence if one of the tasks fails; others continue to the next task. Sequences are the most common task structuring mechanism.

Fork. A fork consists in a point in the workflow model where execution is split into concurrent paths, each of which execute independently from that point onwards.

Join. A join is the opposite of a fork, i.e. a point where several concurrent execution paths are collapsed into one. Some meta-models support only one concurrency construct that aggregates both the behaviour of the fork and that of the join. Moreover, the join construct typically has synchronisation conditions, such as "wait for one" and "wait for all".

Conditional flow. The two most typical ways to support conditional execution of work are the if-like and the switch-like constructions. Either construct causes one of several paths to be taken depending on the result of evaluating a condition that possibly depends on workflow data. The former construct represents an exclusive choice of one out of two exclusive conditions, which cover all the possibilities (e.g. take one of two paths depending on whether some variable x is positive). The latter construct usually has a default path, which is taken if no condition matches. For this reason, it is useful when the conditions don't cover all possible values.

Loop. Loops are iterations over some task or group of tasks. Some meta-models don't have a specific construct for modelling loops whereas most that do impose restrictions to what can be modelled inside a loop.

Block. Tasks are usually the building blocks of a workflow model. Some meta-models go beyond the flat structure and enable modellers to create new tasks by grouping other tasks. This produces a new level of abstraction in the sense that this new task can be manipulated like any other task in the model, thus hiding some details and facilitating the comprehension and creation of the model.

Sub-workflow. The ability to model sub-workflows provides a stronger abstraction and reuse capacity than to simply be able to create nested activities. Support for sub-workflows requires that, at runtime, a new instance of another workflow model (or the same model) be created and executed in its own context. Typical semantics for the invoking workflow are to wait for the sub-workflow to finish (synchronous) or to continue execution (asynchronous) and possibly later get the sub-workflow's output.

Timing constraints. In real life, some tasks need to be executed within a certain time frame. To account for that some meta-models include the ability to declare such constraints in the model. At the same time that a timed task is initiated, a timer is set for the duration of the task. The way that this mechanism affects the flow is when the timer expires; usually some other task must be executed to compensate for the delayed one.

Exception handling. Exceptions are undesired (but expected) or unexpected situations that occur during the execution of a workflow. We only consider whether there is support for the expected exceptions; those whose occurrence is anticipated and to which there is a handling behaviour modelled in the workflow. The other kind of exceptions cannot be in the model (because they are not expected). Some workflow models that don't deal with exceptional behaviour cause the engines to simply fail the workflow execution when such situations occur. Other engines ignore the exception and continue, possibly creating more failures ahead. Meta-models that address this issue usually contain support for the description of tasks that are never executed unless some exceptional situation occurs.

3.1.4 Informational Perspective

Data is the computer representation of the factual information produced by the work. This category deals with the relevant aspects of data representation in the meta-model and its handling between tasks.

Typed data. This characteristic refers to whether it is possible to declare the data types of the data managed. Some meta-models only support a set of predefined data types, whereas others support complex and user-defined types.

Task parameters. We check whether it is possible to parameterise the work on its invocation, by declaring arguments in the workflow model, which are at runtime passed to the work to be executed.

Data flow. Some meta-models may contain only information about the data types used (e.g. to support data persistence) or information about the data produced by each of the tasks, and don't provide a representation for how data flows between tasks, i.e., tasks can receive and produce data but the handling of the data from one task to another may or may not be explicit in the model. When it is not represented in the model, either the WfMS provides a shared space to transfer data or tasks need to depend on an external system to exchange data.

3.1.5 Organisational Perspective

Some meta-models describe the users (humans or systems) within an organisation and permit assigning tasks only to certain users according to some logic. This category evaluates the basic characteristics common to most meta-models that support such representations.

Users and groups. Whether the meta-model supports declaration of users. Groups are simply sets of users, which usually have common characteristics.

Roles. Roles describe some organisational ability or quality. They allow for the separation between users and tasks and for the runtime calculation of assignments.

Task assignment. Whether the meta-model supports assigning users or roles to the execution of tasks.

3.2 jBPM

The jBPM [jBP] is meant to be a flexible and extensible WfMS. jBPM achieves this by having a set of points where the system's default behaviour can be replaced with other implementation code (typically programmer provided extensions). Workflows are modelled in a proprietary language (jPDL). The system is implemented as a set of J2EE components and it has recently become part of the JBoss application server.

A workflow definition is represented using a directed graph with exactly one start node. Each node in the graph is of a specific type. The type of the node defines its runtime behaviour. jBPM has a set of pre-implemented node types. It is possible to write custom code for implementing specific node behaviour.

At runtime, graph navigation is performed using tokens. There is one token for each path of execution. A token maintains a pointer to a node in the graph[2]. A token can change from one node to another when it is signalled and if there is a graph transition connecting the two nodes. The state of a running workflow is given by the position of its tokens.

Node execution starts whenever a token enters the node. Each node must implement the behaviour for the propagation of the token after its execution. There are four possible behaviours for token propagation:

- **Do not propagate the execution**. In this case the node behaves as a wait state. The held token can still be propagated later, thus ending the wait.
- **Propagate the execution over one of the leaving transitions of the node**. The token that originally arrived at the node is passed over one of the leaving transitions of this node.
- **Create new paths of execution**. A node can create new tokens. Each new token represents a new path of execution and each new token can be passed over the node's leaving transitions. An example of this kind of behaviour is the fork node.
- **End paths of execution**. A node can end a path of execution by terminating the token.

Graph execution ends when all tokens have entered a wait state and are not propagated. The node types that jBPM already implements are:

- **Task-node**—it represents one or more tasks that are to be performed by humans. When a task node starts its execution, it creates task instances in the task lists of the workflow participants

[2]When a workflow instance is started one token is created pointing to the start node.

and then behaves as a wait state until the users perform their work. The task completion will signal the token to proceed.[3]

- **State**—this represents a simple wait state with empty behaviour. It is useful if the workflow needs to wait for an external event.
- **Decision**—this node type should be used when the system needs to automatically make a decision during the workflow execution. The decision on which transition to take can be expressed either by using condition elements on the transitions[4] or by providing a `DecisionHandler`, which is a Java class that must calculate the transition to take.
- **Fork**—splits one path of execution into multiple concurrent paths of execution. It creates a new token for each transition that leaves the fork, creating a parent-child relation between the token that arrives in the fork and each of the new tokens.
- **Join**—a join node will end every token that enters that node. The default behaviour assumes that all tokens that arrive at the join are children of the same parent. When all sibling tokens have arrived in the join, the parent token will be propagated over the unique leaving transition. While there are still sibling tokens active, the join will behave as a wait state.
- **Process-state**—executes a sub-workflow. The parent workflow waits until the sub-workflow finishes before propagating the token.
- **Super-state**—groups a set of nodes in a hierarchical fashion. When a token enters a super-state it will be passed to the first node inside the super-state. Super-states can be nested recursively and there can be transitions to/from any node inside/outside the super-state.
- **Node**—generic node used for extending the set of nodes with custom behaviour. It has a custom action associated (a Java class), which is invoked when execution arrives at the node. This action has the responsibility of propagating the execution if it needs to be continued (as in any other node the predefined actions do).

Additionally to the concepts of nodes and transitions, jPDL has the concept of action. An action is a piece of Java code that is invoked upon the occurrence of an event during the workflow execution, such as a token entering a node, a token leaving a node and a transition being taken. The concept of action serves to abstract details from the graph. The rationale is that the graph is usually constructed by a user with the role of process modeller and this user is not interested in the technical details required to the implementation of the workflow. After the workflow is modelled it can be decorated with actions. The main difference between an action placed in an event and an action placed in a node is that the event actions have no influence over the control flow of the workflow whereas the node actions have the responsibility of propagating the execution.

The meta-model supports timers. Timers can be created upon events in the process. When a timer expires, an action can be executed or a transition can be taken.

[3] There are several options for specifying how completion of one task instance affects the continuation of the process, such as to wait for all tasks to finish, to wait for the first to finish, and to continue as soon as the task instances are created.

[4] In this case the first transition for which the condition evaluates to `true` is taken.

Data flow is not modelled in jPDL. Despite this, it supports data exchange between states. For each path of execution (token) there is a shared context where data can be stored (in the form of key/value pairs) and retrieved (given the key). This way, data is implicitly passed from one state to another as long as the states are in the same path of execution. It is possible to pass initial data to the workflow instance when is it started.

The language provides a very simple exception handling mechanism. It supports the declaration of `ExceptionHandlers`, which basically describe the name of a Java exception and the action to be performed if it occurs. The fact that an exception occurs does not have a direct effect on the flow of control. No matter if the exception is caught or not, the execution will proceed from the token's current state. Nevertheless, if the exception is caught then the handling action may put the token in an arbitrary node in the graph, thus indirectly affecting control flow.

jBPM implements a basic organisational model for assigning users to tasks. This model is also described using jPDL and contains the following concepts:

- **User**—represents a single individual in the organisation or a system.
- **Group**—a set of users or groups.
- **Membership**—the role a user has in a group.
- **Permission**—the authorisation given to the user or group.

In a workflow model, states can be associated with a swim-lane. Swim-lanes represent that a certain role (membership) is required for the task to be executed. At runtime, an evaluation is performed to check which users can perform which tasks.

It is a very simple organisational model. jBPM provides an interface that can be implemented, which enables replacing this organisational model with another one provided by an external system.

3.3 OSWorkflow

The OSWorkflow [OSW] developers took a very different approach to the development of a WfMS. OSWorkflow can be considered a "low level" WfMS implementation. Typical concepts present in other systems, such as loops and conditions, must be coded in OSWorkflow using a scripting language. The goal for OSWorkflow is to be an extremely flexible system. This goal originates from the acknowledgement that typical workflow solutions always require a lot of integration code to put in place, when integrating them within an enterprise. OSWorkflow achieves its goal at the cost of being more a state-machine with some workflow concepts and less a full-featured WfMS with many dedicated workflow components. OSWorkflow's meta-model is based on the concept of finite state machine.

States represent a workflow *step*, in which several *actions* are available. For the current step, there may be multiple actions. An action may be set to run automatically or be selected to be run manually through user interaction. Actions contain *results*, which are evaluated after the action completes. Each action has at least one *unconditional result* and zero or more *conditional results*. If multiple

conditional results are specified, the first result for which all conditions are met is executed. If no conditional results are specified, or if no conditions are satisfied, then the unconditional result is executed. A result represents a transition between two steps, so every action implies in itself at least one transition (because of the unconditional result). Results also specify the *status* for the steps that they connect, i.e., when a result is executed it sets the status of the step it's leaving from and the status of the step it's arriving at. The meaning of the status information is irrelevant for the workflow engine and can take any value. However, it can be used in the specification of conditions; these conditions can be used in the conditional results or to restrict which actions can be executed within a step.

If a step contains an automatic action, then that action is immediately executed when entering that step, given there are no restrictions imposed by conditions.[5] If a step does not contain any automatic action then the workflow engine waits until an order is given to choose some action. This is indicated using the engine's API and it is the way how manual work is performed.

Note that automatically executing an action does not imply any execution of domain-specific work, because, actions merely contain transitions to another step (which might be the same). To execute domain-specific work it is necessary to use a *function*.[6] Functions represent atomic units of work that can be performed during a workflow transition and that don't affect the workflow navigation. Functions can be specified either in steps or in actions. Moreover, there are pre-functions and post-functions.

When a function is associated with a step, the function is executed whenever a transition enters or leaves the step. Pre-functions execute before entering the step, whereas post-functions execute after leaving the step. Functions associated with actions have similar behaviour, but applied to actions instead of steps.

Given that the outcome of executing an action is to choose only one of its results, the meta-model concepts presented so far do not support the execution on concurrent work. This must be modelled with two specific states. Forking is done by nesting several results inside a special *split state*. Joining several split paths requires another special *join state*. For the join there must be a boolean expression that indicates the condition under which the join is performed. This condition is a good example of the flexibility in OSWorkflow. Basically any conceivable workflow pattern can be supported by their workflow model. Of course, this comes at the price of having to script almost every bit of behaviour. For example, the simplest "wait for all" synchronisation (AND-join) requires the text shown in the example in Figure 3.1. It is not even a reusable construction, because it explicitly refers which states it is joining.

Branching can be achieved in two ways. The first way is to use a state with one action that has several conditional results. The second way is to use a state with more than one action where each action has one unconditional result. The semantic difference between both solutions is on who has the responsibility for taking the decision of which path to take. In the first case the decision is made by the engine when it evaluates the conditions in each of the results. The first one that matches is taken. In the second case the decision is taken externally when, using the engine's API, someone

[5] If the step contains more than one automatic action that can be executed, then only the first declared action is executed.

[6] Currently, OSWorkflow supports the following function types: Java classes loaded by a ClassLoader, Java classes retrieved via JNDI, and Remote and Local EJBs.

```
<join id="1">
  <conditions type="AND">
    <condition type="beanshell">
      <arg name="script">
        "Finished".equals(jn.getStep(6).getStatus())
        && "Finished".equals(jn.getStep(8).getStatus())
      </arg>
    </condition>
  </conditions>
</join>
```

Figure 3.1: Example of a Join construction in OSWorkflow.

chooses the action to take. In both cases if there are two mutually exclusive conditions we have the *if* behaviour; if not we have the *switch* behaviour.

It is possible, using OSWorkflow's modelling language, to represent arguments that are to be passed to functions. However, it is not possible to explicitly model the flow of data between any two functions. Data can still be exchanged between functions using a shared context, which exists for each workflow instance.

The OSWorkflow does not have, in its meta-model any explicit mechanism for exception handling.

In OSWorkflow the organisational model is even simpler than in jBPM. OSWorkflow's meta-model does not represent roles. It is possible to register the user that is performing an action and it is possible to assign permissions (modelled as boolean conditions) based on the user and the current workflow state.

3.4 Twister

Contrarily to the two previous WfMSs, Twister's [Twi] developers opted to use one of the many proposed workflow standards for their workflow model. Their engine is based on the *Business Process Execution Language* (BPEL) [ACD+03] specification and Web Service standards. Given the nature of the BPEL specification, Twister's real strength should be to perform process orchestration through web service invocations.

BPEL represents a convergence of two earlier workflow languages, *Web Services Flow Language* (WSFL) [Ley01] and XLANG [Tha01]. WSFL was designed by IBM and is based on the concept of directed graphs. XLANG was designed by Microsoft and is a block-structured language. BPEL combines both approaches providing a mix of structured and unstructured representation in the workflow model.

In this model, there are the *basic activities*, which represent the work to be performed and there are the *compound activities*, which represent the control flow. The latter are used to aggregate other activities (either basic or compound) together in a structured form. A special compound activity is

the `flow` activity. This activity is what permits the unstructured control flow as explained further ahead.

The entire model is based on web service invocations. All domain-specific work is implemented in web service operations. The available types of basic activities are:

- **Invoke**—this activity invokes an operation on some web service. Whether it is a synchronous or an asynchronous invocation depends on the operation's description. Input/output operations are synchronous; Input-only operations are asynchronous.
- **Receive**—it is the opposite activity of invoke. It commands workflow execution to wait for an external invocation.
- **Reply**—generates the response of a synchronous input/output operation.
- **Assign**—copies data from one variable to another. It is used to transfer data along the workflow.
- **Throw**—indicates that something went wrong, i.e., a fault or exception occurred in the execution of an operation.
- **Terminate**—immediately terminates the entire workflow instance.
- **Wait**—causes a delay in the execution of the workflow. It is possible to specify either a duration or a deadline.
- **Empty**—this activity does not perform any work.

The compound activities structure the work. They can also be applied recursively. The available compound activities are:

- **Sequence**—establishes an order for the execution of a set of activities.
- **Switch**—creates a decision point using the "switch-case" approach.
- **Pick**—executes one of several alternative branches depending on an event.
- **While**—contains an activity that is executed cyclicly while some condition holds true.
- **Flow**—indicates that a set of activities should be executed in parallel. Within the activities executing in parallel, it is possible to indicate execution order constraints by using *links*.

Both the switch and the pick activities represent decision points but with some differences. The switch activity will only execute the first of its branches that gets evaluated to `true`. A default branch may be defined, which is executed if and only if no other branch was executed. The pick activity will also only execute one branch, but it does not have a default branch. Its branches are of the form event-activity where an event can be either a message or a timer. So, switch is used for decisions driven by workflow data whereas pick is for event-driven decisions. The decisions based on workflow data are the most commonly supported in meta-models, but they are restricted to the context of the executing workflow, whereas event-driven decisions enable the workflow to react to external occurrences.

There are two common uses for external events: One is to produce timers that make sure that a workflow instance proceeds and is not blocked indefinitely waiting for some task to finish; another is to support asynchronous inter-process communication. Without the ability to model event-driven decisions, the alternative to both scenarios would be to model an active polling cycle, which repeatedly checked for the desired state (either the timeout or the arrival of some message). The latter solution typically consumes more resources, because it requires work to be executed several times while waiting on the condition.

The flow activity is the basis for concurrent execution of activities. Any activity directly nested within a flow activity will be executed in concurrency with the others. A flow activity is complete when all the activities of the flow have completed. It further enables expression of synchronisation dependencies between activities that are nested directly or indirectly within it. The *link* construct is used to express these synchronisation dependencies. A link between any two activities states that the source activity needs to finish before the target activity can be executed, independently of any other structuring provided by other compound activities. Each link can have a transition condition, hence unstructured branching can be modelled using the link construct. Links can also cross the boundaries of structured activities. When this happens care must be taken to ensure the intended behaviour of the process.

Another differentiating aspect of this meta-model is the distinction between *executable* and *abstract* processes. Being and orchestration-driven model, its goal is to support both the description of the workflows that are internal to an organisation, as well as workflows than cross the boundaries of organisations. In an executable process it is possible to model the exact details of the workflow. Executable processes can be executed by the engine. An abstract processes specifies the public messages exchanged between parties. It does not include any internal details of flow and is not executable.

Again, because BPEL is purely based on invoking web services for executing the work, it is not possible to model the composition of work that includes human intervention in a process[7].

As opposed to the meta-models presented before, in BPEL we have an explicit data flow model represented together with the control flow. BPEL is all about message passing and web service invocations. So, besides supporting the declaration of data types, parameters and return types used in each activity, it is also possible to model data transfer from one activity's output to another activity's input. This is done with the *assign* activity. More generally, the assign activity copies data (or part of it) from one variable to another.

In BPEL there are two different types of handlers to deal with exceptional situations. There are *fault handlers* and there are *compensation handlers*.

Fault handlers are meant to describe the workflow behaviour when dealing with error situations that cause an activity to fail **during its execution**. Fault handlers work much like exception handlers do in programming languages. Each activity is executed within a scope, which roughly corresponds to its nesting level. For each scope it is possible to define zero or more fault handlers. Whenever an error occurs during the execution of an activity, the list of fault handlers is scanned. If there is

[7]Even though the meta-model does not permit modelling manual work, Twister has a worklist component for human activities. This component is contacted via a web service call using the invoke activity, like any other service. It supports e.g. requesting the creation of a work item and marking its completion.

any fault handler that can handle the fault that occurred, then it is executed. The search starts in the activity's scope. If no fault handler exists that can deal with the fault, then the fault is re-thrown to the containing scope.

Compensation handlers are different because a compensation handler for a scope can only be invoked when the scope **completes its execution normally**.[8] Compensation handlers are meant to be used in situations where nothing has failed explicitly in the workflow, but some "turn of events" required that the already executed activities be semantically undone. An example, is when a company receives an order and before it begins the actual shipping it receives a cancellation; then a compensation activity might be invoked to undo whatever the activity that received the order already did.

As stated before, there is no support in the meta-model for the representation of manual work, because it solely focuses on web services composition. For the same reason, the model does not represent allocation of resources to an activity's execution, nor does it have any access control policies.[9]

3.5 YAWL

YAWL [YAW] is both the name of a language [AH05] and a WfMS [AADH04] implementation. The development of YAWL is based on the very detailed work on workflow patterns [AHKB03, AHKB00] and is an effort to provide a formal foundation for workflow languages. YAWL is inspired in Petri nets, but extends them to provide direct support to most of the workflow patterns the authors have identified.

A workflow model in YAWL, is a set of process definitions organised in a hierarchical structure. A process definition contains tasks, which are either atomic or composite. Each composite task refers to other process definitions at a lower level in the hierarchy. Atomic tasks form the leaves and have no further decomposition. Independently of being atomic or composite, tasks are connected in a graph-like structure to create control flow dependencies. Tasks can be connected directly or through conditions. If tasks are connected through a condition then the condition must be met for the transitions between tasks to occur.

Besides task granularity (atomic or composite), tasks have two other characterising dimensions: multiplicity and split/join behaviour. Regarding multiplicity, it represents whether a task can have only one or multiple executing instances. This is usually used in conjunction with conditions to spawn more than one instance of exactly the same task in a controlled fashion. For split/join behaviour there are three types (for both split and join):

- **AND-split**—all outgoing transitions are taken.
- **XOR-split**—only one outgoing transition is taken.
- **OR-split**—any number of the available outgoing transitions can be taken.

[8] The completion of the activity of a fault handler, even when it does not re-throw the fault handled, is never considered as a successful completion of the attached scope and compensation is never enabled for a scope that has had an associated fault handler invoked.

[9] Twister does, however, have a separate component for handling the concepts of user, group and role, so it is possible to represent allocation of resources to activities although using a separate model.

- **AND-join**—all incoming transitions are expected. Continue only after all incoming transitions are performed.
- **XOR-join**—only one incoming transition is expected. Continue after that.
- **OR-join**—several incoming transitions are expected. Continue after the expected number of transitions is accepted.

Domain-specific work is performed inside the atomic tasks. YAWL is based on web service invocations to perform the work and the meta-model does not differentiate between manual and automatic work[10].

The first version of the YAWL meta-model did not have support for data modelling. Recently, the meta-model evolved and now it includes support both for variables with their data types declaration, and for representation of data flow between tasks. Variables can be declared in the root task or inside a composite task, thus creating only scoped definitions. Data present at any given scope can be mapped to a task's input or from a task's output using an XQuery expression—as long as the task is within that data's scope. One difference between the data flow model supported by BPEL and the one supported by YAWL is that the former uses an activity to explicitly copy data from one variable to another whereas the latter represents the assignment nested inside a task. Even though the practical result, from the workflow designer's point of view, might be the same (i.e., data flows from one step to another), the two methods are fundamentally different in one aspect: In BPEL's method, the act of manipulating data is modelled as a workflow step of its own, i.e., the designer must model an activity whose functionality is to copy data from one place to another. In YAWL's method, each task that produces or consumes some data makes that explicit in the model, i.e. a task produces data and stores it in some variable and another task consumes the data from that variable. If one were to be monitoring the execution of a workflow instance, in YAWL, it would not be possible to separate the begin/end of a task from the act of passing data to another task. In BPEL, this could be achieved.

Regarding the capacity to model exception handling, YAWL provides a mechanism that is based on Petri nets' ability to model implicit choices. Usual exception handling mechanisms contain an explicit declaration stating that when some error occurs then some task should be performed. In YAWL it is possible to model that a group of tokens[11] should be removed from a marked region. The enabling of the task that will perform the cancellation action may depend on the tokens that are within the region to be "cancelled". When the cancelling task completes, all tokens in the marked region are removed.

There is no organisational model in YAWL although its development is planned.

3.6 WfMOpen

The WfMOpen WfMS is a J2EE-based implementation, that adheres to the standards proposed by the WfMC's WfRM [Hol95] and the OMG's Workflow Management Facility [OMG00]. The meta-

[10] Like Twister, YAWL has a worklist component for human interaction. Manual work is not modelled but can be performed using a task that interacts with the worklist component.

[11] Ordinary Petri net tokens are used to represent the state of execution.

model is the XPDL (WfMC's XML Process Definition Language) [WMC02] with some minor extensions. The two most relevant extensions are the support for application declaration and the addition of an attribute to support the *Deferred Choice* workflow pattern [AHKB03]. According to the developers, the former extension was necessary because, while trying to define a language that could be used for any workflow engine, the WfMC had to leave out some information required to bind an application declaration with its implementation. The latter extension is meant to support a workflow pattern that would otherwise require more complex constructions. Both extensions are achieved using vendor-specific extensions, which are accepted in XPDL.

In XPDL the core concept is the *activity*. An activity is an attribute-rich element, because almost all of the workflow information is attached to an activity. It may contain information about implementation alternatives, resource assignment, priority, deadline, and data to be used specifically in simulation situations. In addition, it can describe restrictions to data access and to transition evaluation.

Figure 3.2 presents the structure of the four activity types available in XPDL. All activities have a join and a fork element. The Join specifies how transitions targeting the activity are synchronised. Two types are supported: AND-join and XOR-join. The split element has the same two types and similar semantics, but applied to the outgoing transitions.

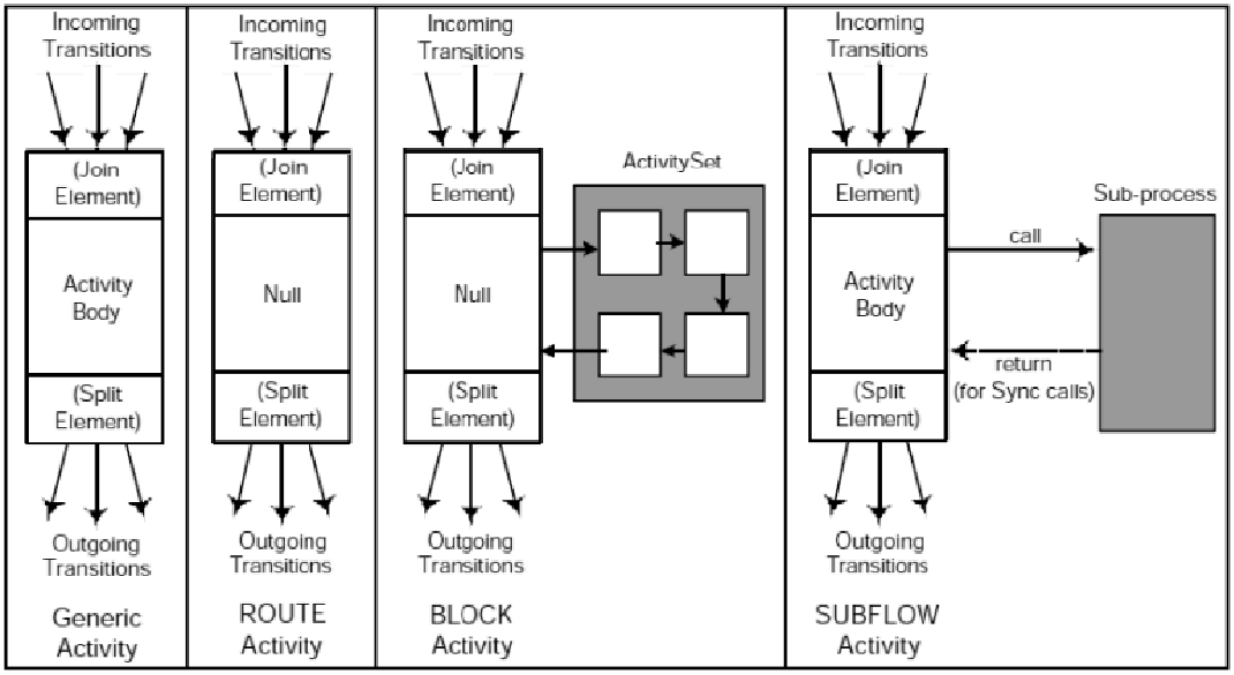

Figure 3.2: XPDL activities.

The *generic* activity type is used to model an atomic unit of work. If an implementation is not provided for the body then it represents an activity that is modelled but whose execution is not enacted by any software component, i.e. it requires human intervention. The *subflow* activity type represents the invocation of another workflow instance, which can be executed either synchronous or asynchronously (i.e. whether the caller waits for the sub-workflow's termination). *Route* activities are "dummy" activities. They can be used to model cascading transition conditions or complex

combinations of XOR and AND split/join conditions. The *block* activity type groups a set of activities together, creating a higher-level abstraction for a part of the workflow. Execution of a block activity waits for the termination of the execution of the activities inside the block before proceeding.

XPDL has an original concept for controlling activity execution. Each activity has an *automation mode*, which is specified for the start and for the end of the activity. There are two modes:

- **Automatic mode**—the engine has full control of the activity's execution. The engine proceeds with the execution of the activity's body automatically, as soon as the join element is satisfied with the incoming transitions. Similarly, completion of the activity and progression to any post activity conditional logic occurs automatically on termination of the invoked application.
- **Manual mode**—the engine requires explicit user interaction to cause activity start or finish. If the start of an activity is manual then after the incoming transition conditions are satisfied the engine waits for a user to accept the execution of the activity's body. Likewise, if the end of an activity is manual then after the execution of the activity's body the engine waits.

Activities are connected through transitions. Transitions have conditions with the usual semantics: A transition is only taken if the condition evaluates to `true`, which is also the default condition when not specified.

Loops can be represented using a transition that returns to an activity that was on a path that led to the transition (circular graph). WfMOpen states that dealing with a loop transition requires that "the workflow engine first resets the state of all activities on the path that lead to that activity to *not started* and then (re-)starts the target activity".[12]

XPDL defines an extensive set of predefined simple and complex data types. It also supports the definition of custom data types using the XML schema notation. The data model describes the data types used in the workflow, the variables and their default initial values, and the applications and their formal parameters. The declared applications are then used to implement the body of generic activities, where the actual parameters are given. The actual parameters refer to variables declared in the scope of the activity. This is how data flow occurs: At some point during the workflow execution an activity outputs information, which is then stored in a variable that is later used as an actual parameter to an application in a following activity. The data flow mechanism used by XPDL keeps the decoupling of applications and activities, minimising the impact of changes in the applications' API (e.g. in the parameters' names). A disadvantage of this solution is that it requires a variable (to act as a container) for each data item that needs to be passed from one activity to another.

Activities also have a *deadline* attribute, which can effectively be used to model exception handling behaviour. A deadline consists in a pair defining the time of the deadline and the exception to raise when the deadline time arrives. The deadline is active while the activity to which the deadline belongs is executing. When an exception is raised it causes another activity to be executed. There are two types of deadlines: Asynchronous deadlines consist of an implicit AND-split operation; the executing activity continues its execution and the exception handling activity is started. Synchronous

[12] We believe that this semantics might not support models in which the looped activities have a mix of complex fork/join rules, where one activity might still be executing while the loop transition is being taken.

deadlines cause the abnormal termination of the executing activity before starting the exception handling activity.

Besides deadlines, there are no other exceptions considered during the execution of an activity, because the meta-model assumes that all activities execute atomically and deal internally with any errors that may occur.

XPDL supports the declaration of *workflow participants*, which are resources that can act as performers in the execution of an activity. In this meta-model, resources can be users, systems, skills, responsibilities, and so on. In the model, resources can be assigned to activities, but during execution the system depends on an external system to provide the semantics of the assignment, i.e., resources declared in the model are logical resources that the external system should map to physical resources.

3.7 DWFMS

The Distributed Workflow Management System (DWFMS) [DWF] implements, as its name implies, a decentralised architecture for workflow execution: It is a major difference from all the WfMSs presented previously. The system is composed by several distributed workflow *habitat*s, which are computational environments for the execution of *software agents*. These software agents can travel from one habitat to another, carrying code and data with them. Once in an habitat, the agents can execute some part of the workflow model using the resources locally available.

The core concept in the meta-model is the concept of *activity*. It represents a unit of work that must be performed by an agent. The implementation of the domain-specific work inside the activity is provided in a *business function*. Business functions can be either automatic or manual. Manual business functions require user interaction whereas automatic do not.

Every activity has one of three types. It is either *standard*, *local process* or *remote process*. Standard activities execute some work in the context of the current workflow instance. Local process and remote process are types of activity that start another workflow instance, respectively in the local or in a remote habitat. In all cases the workflow behaviour is to wait for the completion of the invoked business function.

Control flow dependencies are expressed by connecting nodes in a directed graph. The edges represent the transitions and they can have conditions that must be satisfied for the transition to occur. Besides representing activities, the nodes can also be used to represent split and join behaviour. There are six special nodes, which provide the usual three types of split and join behaviour: `AND`, `OR` and `XOR`. Depending on the conditions associated with the transitions, it is possible to model either concurrent behaviour or exclusive choices.

Activities can consume and produce workflow data. For each business function it is possible to model which are its inputs and outputs in terms of data types. There isn't, however, an explicit data flow model, representing data transfer from one activity to another. From what we understood an activity agent passes its data along to the next agent, i.e., data flows implicitly according to the flow of control.

The meta-model does not have exception handling concepts. However, given that the DWFMS has a monitoring component, it would be possible to do exception handling if we wanted, by monitoring the status of running instances and taking corrective actions. Nevertheless, this is a workaround, because it does not rely on using the workflow model to represent the exception handling behaviour.

One of the issues to which this meta-model has given attention is to the representation of the organisational aspects. Each activity can have, associated with its definition, a set of *capabilities*. The capabilities represent the requirements that must be met by whoever executes the activity. They can be seen more or less like an organisational responsibility. Capabilities can be structured hierarchically. *Resources* represent the available organisational resources, which can be workflow participants (e.g. users) or external resources (typically another system). Resources can also be structured hierarchically. It is possible to associate resources with an activity instead of associating capabilities. The difference is that when a resource is associated with an activity it becomes defined at design time who will perform the activity, whereas when a capability is associated with an activity only at runtime will it be checked who fulfils the required capability.

3.8 Overview

Table 3.1 summarises the characteristics of the WfMSs analysed. It uses the dimensions proposed in Section 3.1.

Whenever some characteristic is marked as *not supported* ($\times$) it does not necessarily mean that it is impossible to achieve that feature in the given WfMS; it only means that such feature is not directly supported by the meta-model, i.e. there are no concepts in the meta-model that directly model such characteristic although it may be achieved by composition of concepts. Therefore, we only mark as *supported* ($\surd$) features that have native support in the meta-model. There are some more ambiguous cases, such as with WfMOpen's exception handling ability. In this case we say partly supported ($\pm$) because the language provides a mechanism to model exceptional flow, but only when it is caused by a deadline. It does not support handling an error thrown by a domain application, in which case it simply ignores it.

The table confirms a fact already stated by other authors, that each system tends to use a different language (either standard or proprietary) as there is yet no agreed-upon standard. We can also see that, although sometimes referring to the same concepts, the core elements on which the languages are based have different names. The structure aggregating the elements also differs with the biggest difference being between block-structured (hierarchical) vs. direct graph (flat) representation. A tree-like representation favours the creation of structured workflows and it is easier to use when the modeller thinks of a workflow as a structured program. However, there is a restriction in hierarchical structures. They do not allow for arbitrary dependencies between steps. BPEL, which uses a hierarchical structure, has a workaround that consists in allowing the `link` construct to cross the boundaries of the hierarchical structure and to create a dependency between any two activities. But, given that workflow modellers tend to think of direct task dependencies the most common structure used is flat.

		WfMS					
		jBPM	OSWorkflow	Twister	YAWL	WfMOpen	DWFMS
Meta-model	Language	Proprietary (jPDL)	Proprietary	Standard (BPEL)	Proprietary (YAWL)	Standard (XPDL)	Proprietary
	Structure	Directed graph	State machine	Tree	High-level Petri net	Directed graph	Directed graph
	Basic elements	State, action and transition	Step, action and function	Basic and compound activity	Atomic and composite task, and condition	Activity and transition	Activity and transition
Functional Perspective	Manual work	√	√	×	×	√	√
	Automatic work	√	√	√	√	√	√
	Domain integration	Java	Java	Web services	Web services	JavaScript, Java, Web Services, Jelly, LDAP	Java
Behavioural Perspective	Sequence	√	√	√	√	√	√
	Fork	√	√	√	√	√	√
	Join	√	±	√	√	√	√
	Conditional flow	√	√	√	√	√	√
	Loop	×	×	√	×	×	×
	Block	√	×	√	√	√	×
	Sub-workflow	√	×	×	×	√	√
	Timing constraints	√	×	√	×	√	√
	Exception handling	√	×	√	±	±	×
Informational Perspective	Data types	×	×	√	√	√	√
	Task parameters	±	√	√	√	√	√
	Data flow	×	×	√	√	√	±
Organisational Perspective	Users/groups	√	√	×	×	√	√
	Roles	√	×	×	×	±	√
	Task assignment	√	√	×	×	√	√

Legend: √ = supported, ± = partly supported, × = not supported.

Table 3.1: Overview of WfMS's characteristics.

Regarding the representation of the domain-specific work, it is worth noting some facts, as well. The most interesting is perhaps whether the meta-model provides direct support for manual work. Again, meta-models that don't support it, are not hindered from interacting with human workers; they simply require a wrapper application that interacts with some worklist component. Hence, from the modeller's point of view, it's always about orchestrating software components. The disadvantage of these meta-models is that the information of who performs the work (human/system) tends to be lost from the model.

In the context of manual vs. automatic work, XPDL provides a unique concept around the execution of each activity. Besides representing manual and automatic work (depending on whether the body of the activity is provided), every activity has an automation mode associated with the start and the end of its execution. The idea is to allow for control over the enactment of all activities, e.g., it is possible to have manual control over the beginning/ending of a fully automatic activity. This feature can be very useful, for example, when creating and executing incomplete workflow models. Consider a very simple workflow with four well-defined sequential steps, where there is some business logic between the second and the third steps that is not yet modelled and can only be known at runtime. The second step might have a manual end which effectively holds the workflow's execution until some user marks it as finished. While the workflow is waiting for such termination the non-

modelled part can be executed; or in the presence of an adaptive workflow system the missing part could be integrated into the model just before it became necessary and then executed as an integral part of the model.

All of the evaluated WfMSs are Java-based, which obviously gives them the tendency to support Java as the domain integration language. Some go beyond that and provide web services support in the models. Either way, all these WfMSs are easily extended with other domain integration features (of course, this requires programming).

As it would be expected all the meta-models provide support for the most common control flow primitives (sequencing, branching, and concurrent execution). The exception is OSWorkflow. We consider that joining is only partly supported in OSWorkflow because it is always necessary to code the joining semantics. OSWorkflow's meta-model is very powerful, but also very complex to use. Its power comes from the possibility to script much of the behaviour such as the joins. Given that the scripting language is directly a part of the model it is very easy to express custom behaviour. The price to pay is that much of the model's behaviour is hidden inside the code and is not easily extracted by a model analysis tool.

The direct support for loops is almost absent. BPEL is the only one to support it natively through the structured construct `while`. In fact, loops are either not supported at all (not even by construction) or supported under some restrictions (such as all behaviour inside a loop must be finished before the control flow loops back).

Let's analyse why there are so many problems with loops. First of all, structured loops are easier to model and to implement, because it is clearer to see which tasks are being iterated and how many loops there are (the same is not true for arbitrary cycles). However, the biggest problem with structured loops is that they limit the control flow possibilities. It is not possible to jump back to an arbitrary point in the model. Most workflow designers believe that arbitrary cycles are closer to modelling real life workflows, which is why most meta-models that support loop execution do so by allowing transitions to go anywhere in the model and do not impose a structured cycle[13]. This form of representing loops poses another problem, because it uses "normal" sequential dependencies that do not identify explicitly that they are creating a loop. Hence, it is only possible to identify the loop by doing a model analysis and checking that the transition is pointing to an already executed task. However, when there are many entangled arbitrary cycles it becomes very difficult to assert where are the loop transitions and where are the "normal" transitions.

Therefore we end up with the three most common solutions regarding loop support: a) the meta-model supports structured loops; b) the meta-model supports unstructured loops, but imposes restrictions to what can be executed; c) the meta-model does not support loops and iterative behaviour must be implemented outside the workflow model (e.g. using domain-specific code). The second alternative is the most common.

Support for blocks and sub-workflows serves a common purpose: The ability to create a structure that abstracts part of the model. The difference is on how each one is meant to be used. A block is a definition that is local to a single workflow model, whereas a sub-workflow defines a

[13] BPEL also supports arbitrary cycles with the `flow` construct.

reusable and independent workflow model. Within a block it is usually possible to access global (or higher-level) structures of the workflow instance to which the block belongs. The same is not true in a sub-workflow, which only accesses data that is explicitly handed to it and is not aware of higher-level contexts. Different WfMSs give blocks different uses. For example, in jBPM the main reason for using blocks is to set events that run whenever entering and leaving the block's execution; in WfMOpen blocks are used to define data scopes and create self-contained parts of the workflow model; in Twister, blocks provide the structure for modelling the control flow. The main reason for using sub-workflows is usually the same: Design reuse. Moreover, when invoking a sub-workflow the meta-models provide two execution policies: Synchronous and asynchronous execution. In the former policy, the invoker waits for the sub-workflow to finish before continuing, whereas in the latter policy it does not. In the second case, merging of the concurrent executions is generally supported using a task that waits for an external event. It is trivial to support synchronous sub-workflows in meta-models that do not have sub-workflow concepts, simply by programming an automatic domain-specific task that launches a workflow instance using the engine's API. This task waits for the sub-workflow's end before finishing itself. It also passes and retrieves the necessary data. Asynchronous implementation is similar but it requires another domain-specific task if we intend to synchronise the two workflows later on, because after starting the sub-workflow, the invoker continues its execution.

Note that even if the meta-model supports the sub-workflow concept, the solution of using domain-specific tasks must still be used if the sub-workflow we wish to invoke is represented in another language. In this case, the problem is that the sub-workflow most probably needs to be started in another engine, because each engine only supports one language.

Exception handling is a complex issue. Some meta-models simply ignore it, whereas the others that deal with it have different levels of support. There are meta-models, such as jPDL and BPEL, which accept that the execution of the domain-specific work can fail, thus producing some kind of exception in the model's execution. The difficulty in handling such cases, is that it is generally very hard to model a "good" workflow that deals with the failure, because it occurred in the domain-specific work, whose behaviour is not specified in the model. Depending on the failure, workflow execution might be recoverable or not. Common solutions are to abort the workflow execution, to invoke another domain-specific task that will try to correct/undo/compensate the failure, or more generally to diverge the workflow execution to another point. Meta-models such as YAWL's, provide a different exception handling mechanism. Instead of executing some task when an exceptional circumstance occurs, in YAWL the execution of some task(s) is cancelled when another task is started. The implication is that the decision to start the exception handling path must be taken, either by a workflow user or by an automatic step. Another meta-model, XPDL considers that the domain activities deal internally with any problems they may encounter and therefore the meta-model does not consider such failures. The closest thing to exception handling, in XPDL, is the use of a deadline event to start a given activity. The disadvantage is that it can only address problems that can be assessed on a temporal basis.

Regarding the data model category, we can observe that YAWL and the proposed standards, provide full support for all the characteristics considered. However, even if indirectly (i.e. not in the

model), all of the analysed WfMSs have a way of handling data. For example, in the solutions provided by jBPM and OSWorkflow, data doesn't flow; it is kept in a shared space that each activity can access to store and retrieve data. Moreover, the data types are not described in the models and any object can be exchanged between steps.[14]. In DWFMS, only the explicit declaration of data flows is missing, but given that data is passed implicitly along with the agents, this issue is solved differently. Still, this approach requires that there is a control flow dependency between agents so that data can pass from one to the other.

Perhaps, the biggest advantage of having the data types declared in the model is that it is then possible to have a mapping mechanism for the sub-elements of the data type. For example, consider two activities `A` and `B`: Activity `A` outputs a phone number used by `B`. In fact, `A` outputs a `Person`, which contains a `Phone` sub-element. This sub-element is the type of data that `B` needs. If the meta-model contains the description of the data types then it is possible to model that only the `Phone` data is to be given to `B`. Otherwise, `B` would have to know about the `Person` data type just to extract a phone number from it.[15]

Organisational concepts are not much developed in the meta-models yet (even in those that already provide task/resource assignment). They mostly provide only a small subset of all the organisational concepts. This subset is usually sufficient for the kind of information a WfMS needs to have when executing a workflow. Resource management is generally left to an external system that accesses a database of organisational information. The most common operation during workflow enactment is to assign work to some resource and, to do that, the system might check which resources have the necessary privileges or abilities to perform the work.

The meta-models that don't provide any support for the organisational model (BPEL and YAWL), are also the same that don't support representation of manual work. Typically, these meta-models are concerned only with connecting software applications, in which case the organisational information is less relevant. On the other side of the spectrum is the organisational model for DWFMS. Even though it is simple, it is the most expressive one, mainly because of its hierarchical decomposition and the possibility of associating either capabilities or resources to an activity. Perhaps, because it is a distributed system based on agents, the need for describing the organisational resources for the enactment of the workflow, became more relevant than in other systems.

In the overall, WfMOpen's meta-model is the one that better fulfils all the categories analysed. It is also based on the most commonly used structure for workflow modelling—directed graphs. YAWL's greatest quality is the solid theoretical base on the workflow patterns study. The inclusion of an organisation model is planned although it is currently lacking. On the contrary, DWFMS has a good organisational model. Being a distributed system, it has other qualities not evaluated in this study, such as better scalability. Still, the workflow definitions are created in a centralised fashion. OSWorkflow is very good as a framework, but not as a final product, which even the authors acknowledge. Twister has a strong standard (perhaps the strongest at the moment) as an influencing factor, favouring its use, but it requires a separate organisational model. It contains no support for re-

[14]There are, however, restrictions to what data types can be stored persistently.

[15]An alternative solution is to insert a step between `A` and `B`, which takes care of the transformation, but again, this additional complexity would not be necessary if the meta-model already supported it.

source management whatsoever. Last, but not least, jBPM has been much praised by the community, but it lacks a good data model.

3.9 Conclusion

We presented six WfMSs currently under active development. We have consulted other systems, as well, although not all. We believe that the presented systems cover a large spectrum of the most common approaches to WfMS development and that the observations given here also apply to many other systems.

In order to have a uniform evaluation, we needed to have some comparison criteria. These were provided by the framework we presented in Section 3.1. The framework was divided into categories, each of which contained several evaluation dimensions. They were derived both from the WfRM and from other evaluations previously mentioned. We understand that, probably, each of the categories could originate a separate study of its own, but that would be a level of detail out of the scope of this evaluation: Our goal was to give an overall view of the meta-models of current systems.

As a final summary, we draw some general conclusions that affect the workflow field in general, but with emphasis on the meta-models:

1. There are, currently, many WfMSs. Each, usually supports only one meta-model.

2. Regarding the representation of the elements of a workflow, the meta-models differ in structure, as well as in the basic elements they use to represent the same core workflow concepts.

3. There is no agreed-upon standard for a workflow meta-model, not even for what kind of information should be modelled (e.g. should there be an organisational model? Should manual work be modelled?). Nevertheless, there are some implicit agreements, e.g. all meta-models represent work and control flow.

4. Workflow definitions are not currently interchangeable. It is not easy for an organisation that already as a WfMS in place to execute a workflow model written in another format. The common solution is to remake the model in the supported language.[16]

5. There is no mention of an upcoming winner in the standards race. Also, there is no *de facto* standard being adopted. Although the standards proposals attempted to address many details, that did not help them, perhaps because the models they produce are always very big, and complex to understand and manage.

[16] With the creation of XPDL, one of the goals of the WfMC was to create a workflow interchange language, which is no more that another standard proposal that suffered the same faith as others.

4 The Workflow Virtual Machine

In this chapter we propose a model for the WfES. The proposed model addresses the conflict of forces, which we already mentioned, between workflow modellers and WfMS's developers. From the standpoint of a workflow modeller, the modelling language should be easy to use (i.e., user-friendly), preferably supported by modelling tools, contain expressive constructs to help on the modelling process, and designed to help the user to avoid making errors. On the other hand, the developers are concerned with qualities such as extensibility, maintainability and simplicity of implementation—therefore, the language should be minimal, with no redundancy in it.

The root cause of these conflicting forces is the fact that the language that is used to model workflows is the same language that is used to execute them. Most of the time, the need for a change in the language is incited by the modellers, because they want to have more expressiveness, but the developers might also propose changes that simplify their implementation. Either way, anytime a change in the language is necessary, both sides are affected: The modellers need to understand new concepts; the developers need to update the system's implementation.

The basic rationale of our solution is straightforward: We separate the two languages. The model we propose for the core of a WfMS is depicted in Figure 4.1. This model decouples the language used to model workflows from the language used to execute them. It divides the core of the WfMS into two layers:

- A layer implementing a *Workflow Virtual Machine* (WfVM), which is responsible for the WfMS's core activities and has its own workflow language.
- A layer where several WfDLs can be supported.

Despite the existence of many WfDLs, we can observe that they all share a common subset of concepts such as "activity", "control flow", "data flow", and "invocation of domain-specific logic" [Hol95]. The idea is that the WfVM should support the core workflow concepts, and allow the definition and the execution of workflows described in a simple language that uses those concepts. We call this language the *back-end language*. The back-end language is not the language in which WfMS users describe their workflows.[1] That other language—the WfDL for the modellers—is what we call a *front-end language* and is not part of the WfVM.

With a WfVM in place, the description of a workflow in a front-end language is first compiled into a workflow described in the back-end language and only then executed. This way, the core of the WfMS does not depend on the front-end language, which can change freely—provided that it can be compiled to an appropriate back-end description. By shifting the effort of supporting a front-end

[1] Although it could be, but that's not the aim of the back-end language.

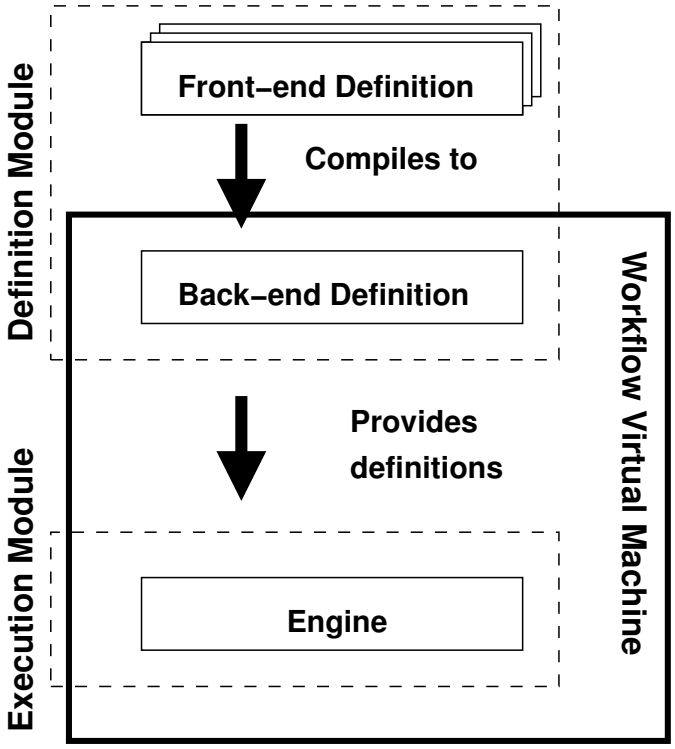

Figure 4.1: The core of the workflow management system.

language to the compilation process only, we have gained a reusable engine, that does not have to be constantly adapted to changes in the language.

The back-end language is not to be manipulated directly by humans, so it does not have the user-friendliness requirements; it can meet the requirements of the developers. The language to be used by humans (the front-end language), is not to be directly operated by the WfVM; it can be as large and user-friendly as necessary. Note that the definition component is split among the two layers: The front-end definition module is not part of the WfVM. The compilation process is the "glue" between the two layers.

As a bonus, it becomes possible to have one instance of a running WfMS that can simultaneously enact workflows written in different languages. We believe that a universal WfDL that suits all users is not feasible. For particular domains, some constructs related to the domain can greatly simplify the modelling task: Different domains have different needs. So, being able to support different front-end languages is an important requirement for a successful WfMS.

In the following sections we describe in detail the two elements of the WfVM: The back-end definition and the execution modules.

4.1 Back-end Definition Module

The back-end definition module provides the WfVM's meta-model for representation of workflow definitions. This module must keep track of the known workflow definitions and provide information about them to the engine, which is the execution module's main element.

4.1.1 Meta-model

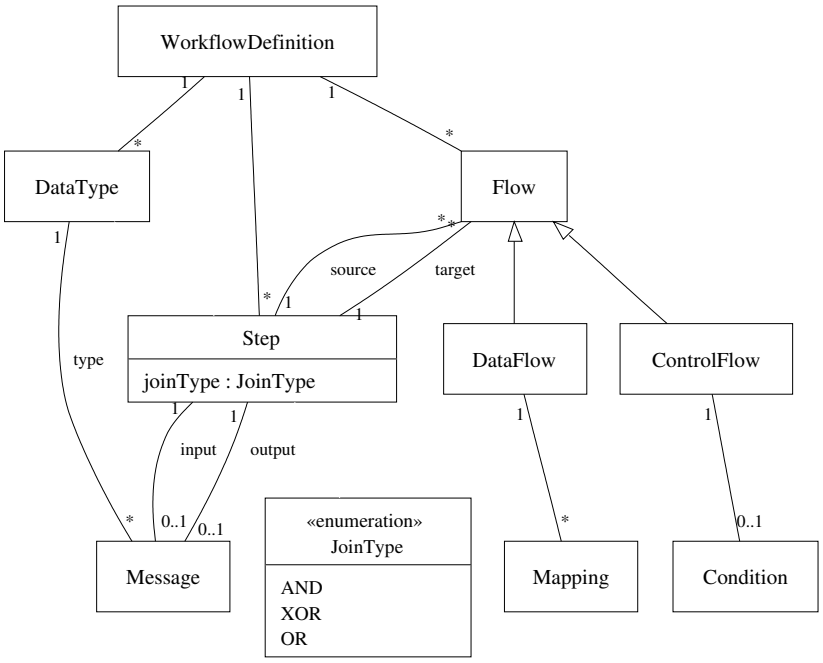

Figure 4.2: Meta-model for the back-end workflow definition.

Figure 4.2 presents the meta-model for the back-end workflow definition. Instances of this model represent workflows that can be enacted by the WfVM's execution module.

A *workflow definition* is the root element. It holds information about the work, the flow, and the description of the data structures handled by the workflow definition that it represents.

A *step* models a point of the workflow definition where something has to be done. Every workflow definition must have a *start* and an *end* step. From the workflow engine's point of view a step represents a task that is performed atomically within the context of the given workflow. Steps can be used to do virtually anything. They will surely be used to model work from the front-end workflow model, but they can also be used to perform any other controlling activities that may arise, for example, from the compilation process.

Each step can optionally have one *input message* and one *output message*. A *message* is a slot for keeping workflow data. It can have an arbitrarily complex structure, which is described by its *type* attribute. The type must be one of the *data types* associated with the workflow definition.

A *flow* represents a transition between two steps. There are two types of flow with different meanings: *Control flow* is for controlling the order of execution of the workflow and *data flow* is for passing data between workflow steps.

A *control flow* dictates the order of execution of the steps and it may also contain a *condition*, which further limits whether the flow can be navigated.[2] The semantics is the following: Given two steps `A` and `B`, and one control flow with condition `c` from `A` to `B`, then `B` can only be executed after `A` finishes and if `c` evaluates to `true`. The previous description considers a simple situation where there is only

[2]If not specified, the condition defaults to `true`.

one control flow targeting `B`, but a step can be the target of many control flows. To manage what happens when multiple control flows target the same step we use the step's synchronisation rule. Each step has one of the following synchronisation rules:

- AND-join: This is the default synchronisation rule. It means that any control flow that targets this step must have been taken before this step begins its execution.

- XOR-join: This step's execution is initiated after the first control flow reaches it. All other incoming control flows are ignored.

- OR-join: Whenever a control flow that targets this step is performed, the execution of this step is initiated, i.e., this step is executed each time an incoming control flow occurs.

Thus, the synchronisation rule is irrelevant when there is only one incoming control flow, because, in such case, the target step of a control flow always begins its execution.

Contrarily to the control flow, a *data flow* does not impose restrictions to the step's execution order. A data flow represents that some data (if available) must be passed from one step to another. Given two steps `C` and `D`, if there is a data flow from step `C` to step `D`, then when `D` starts its execution, the data from `C` is provided to `D`. It is not required that `C` has finished before `D` can begin its execution; of course that, if data is not available, then it cannot be given to `D`. Still, `D` will be executed.

A message can have a complex structure, which is described by its type. It is possible that the data types of source and target messages mismatch. In such cases one must provide *mappings* from the source message to the target message. One mapping indicates which part of the source message should be copied onto which part of the target message. We will further address this mechanism when we discuss the informational perspective.

4.1.2 Functional Perspective

This perspective is directly supported by the concept of step. From the back-end meta-model's point of view there is no further decomposition. Also, when the engine is executing a workflow instance and it schedules the execution of some step, it has no knowledge of what the step is about to do.

If there was no front-end language and if the workflows were directly modelled in this language, then the implementation provided for the step would simply correspond to the execution of the domain-specific work. But given that a front-end language might provide concepts that are not present in the back-end, the step's implementation is usually a wrapper that implements some concept available in the front-end. This separation is necessary so that the WfVM can be used for whatever front-end definition we decide do have. Part of the compilation process from a front-end language to the back-end is to provide the step's behaviour and thus implement, in the back-end, the front-end concepts than cannot be natively modelled (such as manual work).

4.1.3 Behavioural Perspective

The concept of control flow represents all the dependencies between the steps. As stated before, the data flow does not influence the execution order of the steps. In this section we address some relevant aspects about the basic control flow constructions, namely branching and loops. Chapter 6 performs a detailed analysis of the expressive power of the back-end language with focus on the behavioural perspective.

The condition associated with the control flow supports a runtime decision that depends on workflow data. Whenever a step ends, all its outgoing control flow conditions are evaluated. Those that return `true` are taken. The order of the evaluation of the control flows is not specified. If the conditions are mutually exclusive, then only one path will be taken and we have an *if*-like behaviour. If more than one condition evaluates to `true`, then we have the execution of concurrent work.

Concurrent work can be synchronised by having several control flows targeting the same step. The behaviour depends on the type of synchronisation rule of the target step. Unlike other meta-models, there are no restrictions to the splitting and joining execution paths.[3]

Workflows often require iteration over some steps. This is modelled in the back-end by establishing a control flow from one step to another (a backward step), creating a cyclic dependency. All control flows in a graph have to be tagged, indicating whether they are *loop control flows*.[4] Identifying which control flows are loop control flows is important, so that the engine can easily and correctly solve synchronisation rules on steps which are preceded by steps which in turn they also precede. More on this is discussed in the execution module in Section 4.2.

4.1.4 Informational Perspective

Each step can have one input and one output message. They work as the slots for storing the data that the step handles. Each message is typified; its data type contains the description of the structure of a message. Data type descriptions must provide the names for the data fields and their types.

A data flow connects two steps. By default, i.e. if there are no mappings associated, it represents a copy from the source step's output message to the target step's input message, field by field, matching field names. This is the default behaviour because it's the most common and straightforward one. Sometimes, however, we may want to change this behaviour for the following reasons:

- We do not want to copy the whole output data, because the input of the target step takes less fields or, even if it takes the same fields, because they're not all coming from the same source step.
- We need to change the fields' names, because the two steps were created independently and whoever defined their messages used different names.
- We do not want to use the output message of the source step as the data source or we do not

[3] E.g., in some meta-models it is only possible to synchronise parallel execution paths that were created at the same point in the workflow.

[4] By default a control flow is a non-loop control flow.

want to use the input message of the target step as the data target (i.e. we do not want the default behaviour). Note that each step can have two messages declared (one input and one output). Hence, it is possible to choose either one of them as the source or as the target of the data flow.

The mapping concept provides the solution for the requirements stated above. Each data flow can have any number of mappings. All the mappings of a given data flow refer to the same source and target steps (which is already defined by the data flow). For each end of the data flow the mapping specifies:

- Which message to use, i.e. whether it is the step's input or output message.
- Which part to read or write, depending on whether it is the source end or the target end, respectively.

To better illustrate the reason why data flows should not impose restrictions on the order of execution of the steps, consider the example in Figure 4.3. The arrows with solid lines represent control flows (they are loop control flows if the tip of the arrow is hollow) and the arrows with dashed lines represent data flows. The figure depicts an iterative execution between step `N2` and step `N3`. When `N1` finishes its execution, control flow progresses to `N2`. Note, that there are two data flows targeting step `N2`. However, only the data flow from `N1` to `N2` has data to provide, because `N3` hasn't produced any yet. After the first iteration when `N3` finishes and assuming condition `!c == true`, then `N2` is executed again, but this time with the data from `N3`.

If the data flows caused an implicit control flow dependence, then there would be a deadlock at `N2`, because the workflow would wait forever on one of the two data flows that target `N2`.

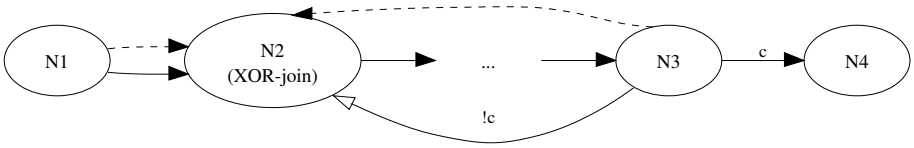

Figure 4.3: Data flow in loop.

4.1.5 Organisational Perspective

In different literature, as well as in the implementation of different WfMSs, opinions about the scope of the organisational perspective differ greatly. Whereas some systems and publications ignore this perspective as a component of workflow management others discuss it in detail and state complex requirements on it.

We believe that the importance of the organisational perspective in WfMSs depends on the application domain. It appears to be less relevant in web service-based workflow languages (it is not even addressed by standards like BPEL [ACD+03]). In WfMSs that distribute work among human users opinions differ whether the organisational perspective belongs within the WfMS or in an external component [DS99, Hol95]. Some of them stress the organisational aspect strongly and provide

sophisticated modelling constructs and policies; others provide minimal support or do not consider the aspect at all.

The WfVM is meant to contain the minimum set of concepts that make it usable with several front-end languages and also to be simple, maintainable and extendable from the programmers' point of view. This means that we have opposite forces: On the one side if the WfVM contains organisational concepts, then it becomes better suited for supporting the front-end meta-models that have such concepts. On the other side, they are not strictly necessary at the core of the system. We have chosen not to include the organisational perspective in the core part of the system.[5]

4.2 Execution Module

The execution module contains the life-cycle management operations for the workflow instances. It is responsible for reading the shared workflow definitions from the definition module and for creating a unique context for the enactment of each workflow instance. It also creates a *step instance* for each step that is scheduled for execution. This way, the WfVM can concurrently execute multiple workflows sharing the same definition. Note that the execution module only interacts with the back-end sub-element of the definition module. The compilation process is what provides the transformation between the language used by the modellers (front-end) and the language used by the engine (back-end).

The most important aspect of the execution module is how its engine enacts a workflow instance, i.e., given the creation of a workflow instance, how is the workflow definition navigated. Thus, the execution algorithm is the focus of this entire section. We split the presentation of the algorithm in three parts: We start by presenting the core of the algorithm stripped from some details. This allows the reader to grasp the main concepts relevant to the enactment of the back-end definition, without cluttering the presentation. Following, we identify some workflow constructions (namely using loops and concurrent work) that still need to be supported, and discuss the implications on the algorithm. Finally, we extend the algorithm with the necessary elements to resolve the issues identified.

This section ends with the presentation of some examples in which we demonstrate the application of the algorithm.

4.2.1 Core Execution Algorithm

We describe the execution algorithm according to each of the workflow perspectives supported by the WfVM. For the functional perspective we describe how a step is executed, for the behavioural perspective we describe how control flows between steps, and for the informational perspective we describe how data flows between steps.

[5] Meanwhile, there has been an entire PhD thesis [Dom05] developed around the subject of extending the core of this WfVM with access control, which in part involved the creation of an organisational model.

Functional Perspective

Given that a step represents a task that is performed atomically within the context of the given workflow instance, this means that there is no further decomposition and that the engine does not have to understand the internals of a step. The only concern is about the separation of the model from the instance. Therefore, whenever a step is scheduled for execution (c.f. behavioural perspective) the engine creates a step instance and places it on the scheduler's queue. The creation of a step instance decouples the model from its instantiation. The execution of a step instance consists simply in the invocation of the domain-specific behaviour implemented in that step during compilation.

Behavioural Perspective

The navigation of the execution graph comprehends three distinct situations: Starting a workflow instance, computing the following step after a given step finishes, and ending a workflow instance.

Remember that every workflow definition must have a start step and an end step. Thus, starting the execution of a workflow instance consists in marking for execution the start step of the corresponding workflow definition (like any other step, this one is executed as described in the functional perspective).

From here on, the navigation takes place whenever a step instance finishes its execution. This is an asynchronous occurrence, i.e., at any given moment one step instance can finish its execution and fire the continuation of the model's navigation. When a step instance finishes, the engine must do the following:

1. Check if it is a end step. If so, end this workflow instance. Otherwise, continue to 2.

2. Iterate through the outgoing control flows from the finished step. For each control flow evaluate its condition. If the condition returns true then do as described in 3. Otherwise, skip to the next control flow.

3. Check if the target step of the control flow can be scheduled for execution. If the target step has a single incoming control flow, then it can be scheduled; otherwise depending on the synchronisation rule:

   - If the target step is an AND-join, then all its incoming control flows must have been traversed already in order to proceed (i.e., the last transition to the step has just been made).
   - If the target step is an XOR-join, then execution will not proceed, unless this is the first transition to the step—this causes all other transitions to the target step to have no effect and allows execution to proceed immediately after the first control flow.
   - If the target step is an OR-join, then the step can be scheduled for execution.

4. Depending on the result of the previous evaluation, decide whether to schedule the target step for execution.

Ending a workflow instance occurs when the step marked as the end step finishes its execution. In this case the engine must mark the workflow instance as finished and terminate any running step instances for this workflow instance.

Informational Perspective

This part of the algorithm deals with handling data from one step to another. Each step can have one input and one output message. Step instances use these messages as the slots for storing data that is received and produced by the step's execution.

Data flows are processed whenever a step instance is about to start and after it finishes its execution. **Before starting** the execution of a step instance the engine applies any data flows that target the step, i.e., it populates the target message with the contents of the source message applying any existing mappings. **After finishing** the execution of a step instance the engine keeps the messages if there are any data flows whose source step is this one (this data will be provided later to some other step).

4.2.2 Adding Requirements

We simplified the description of the execution algorithm, in order to make its initial presentation clearer. The algorithm presented in the previous section is enough to execute workflows that either have a single flow of control or, in the presence of concurrent work, they do not include loops. In the following we provide examples of the situations that mix loops and concurrent work in order to clarify the problems that they present. Based on the cases shown, we derive the requirements to be satisfied by the final (extended) execution algorithm.

First Case: An XOR-join nested in a loop

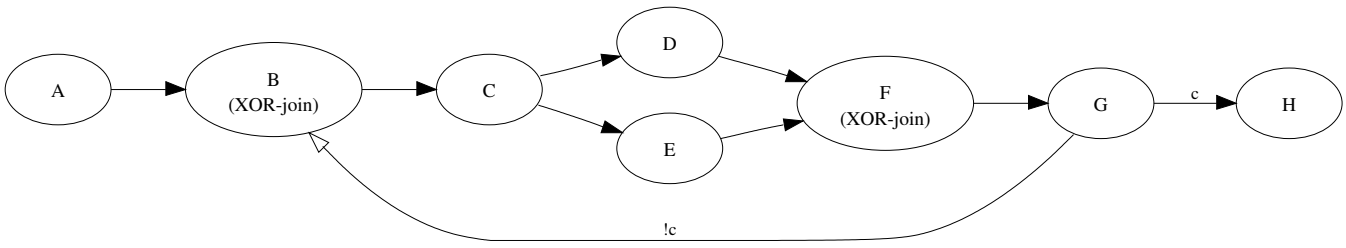

Figure 4.4: Loop with nested split/XOR-join.

Consider the example given in Figure 4.4. Suppose that, at some point during the execution of this workflow, steps `D` and `E` are under execution. After one of them finishes, let's assume `D`, the execution can proceed to `F` (because of the XOR-join rule). Also note that when `E` finishes, its control flow to `F` should not start another execution of `F` (however, in our example, `E` is still executing). After a while, when `G` finishes its execution, if condition `!c == true` is satisfied, then `B` will be executed again; so will `D` and `E`, eventually. Suppose that this time `E` finishes before `D` does. How can the engine know whether `F` should be executed? There are two options:

- If the instance of `E` that finished corresponds to the first iteration, then `F` **should not** be executed.
- If the instance of `E` that finished corresponds to the second iteration, then `F` **should** be executed.

Therefore, **the join step needs to differentiate between multiple transitions of the same control flow**. The synchronisation rules mentioned in the core algorithm apply only to joining paths that correspond to paths split in the same iteration. It is, thus, necessary to include some workflow control data that allows the engine to differentiate between executions of the same flow in different iterations.

A possible solution could be to add an iteration counter that would be passed along the workflow execution. When the workflow was started, the engine would initialise the counter. Each step instance created would keep the counter and whenever a loop control flow was taken, the counter for the target step instance would be incremented. With this simple mechanism the engine could distinguish between two instances of the same step, because they would have different iteration counters. Although this simple solution appears to work, it is not enough to address all the issues with loops and concurrent work, as we shall see in the next cases.

Second Case: Merging branches that contain loops

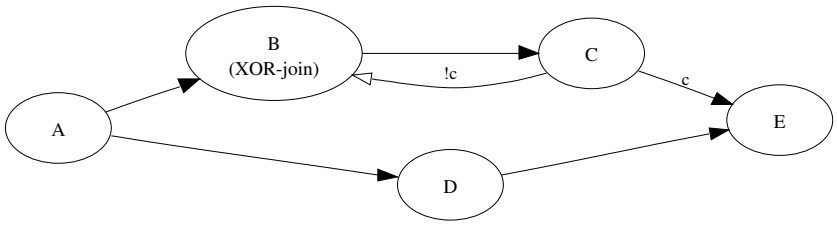

Figure 4.5: Concurrent work containing a loop in one branch.

In the model shown in Figure 4.5 we want to allow the execution of step `E` only after both branches finish (remember the default AND-join rule). The bottom branch will finish as soon as the first (and only) instance of step `D` finishes, but the top branch might perform several iterations of `B` and `C` before `c == true`.

The problem we solved in our first case unveiled another one. In the first case we needed to distinguish step instances, and to solve that, we proposed passing a counter in each step instance, incrementing it for every loop flow taken. If we keep using the same solution in this case, when the loop is executed more than once, the engine cannot recognise the two transitions to step `E` as belonging to the same group, because the counters will be different. **The execution of structured loops should not have any influence outside of the loop's body**.

So, besides incrementing the iteration counter when taking a loop transition, we must also exit a cyclic area with the same counter as we entered it with. This can be done by storing the current iteration counter when entering a loop, which can be identified by entering a step that is the target of a loop transition. In our example, when creating an instance of step `B` for the first time, the engine must store the current iteration counter. When leaving step `C` with `c == true` the engine propagates not `C`'s current iteration counter, but the stored value. This way, `E` will be executed when both branches finish.

Third Case: Nested loops and concurrent work

With this third case we want to show that the generation of iteration counters cannot be done by simply incrementing the current counter when taking a loop transition. Doing so, could lead to erroneous behaviour.

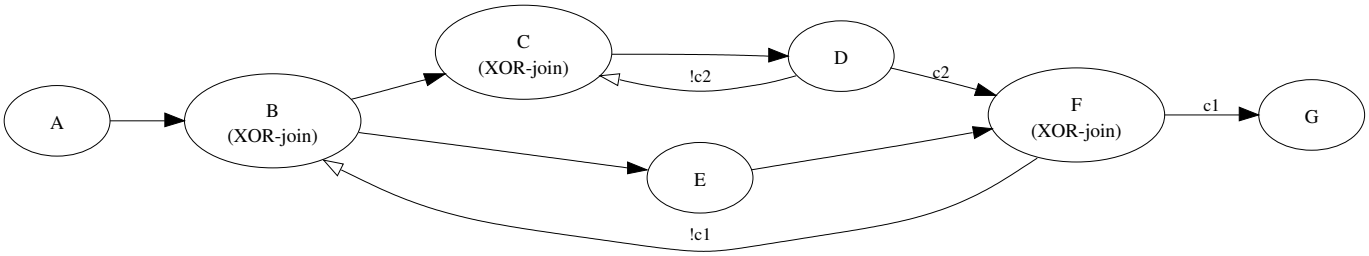

Figure 4.6: Nested loops and concurrent work.

The model in Figure 4.6 has an outer loop (from `B` to `F`), whose body contains concurrent steps. One of the branches is another loop (from `C` to `D`). Notice that step `F` has an XOR-join synchronisation rule. For this reason it is possible that the outer loop is executed again, while one of the branches of its body is still executing in the previous iteration.

Consider this execution sequence: `A`, `B`, `C`, `D`, `C`, `E`, `F`, `B`. The execution of `F` was possible, because `E` finished. However, the top branch is still executing the inner loop. Supposing that when the workflow instance began the counter was set, e.g. to 1, then when traversing from `F` to `B` the counter is incremented to 2. The same thing happened when traversing from `D` to `C`. Now that the outer loop's body will be executed again, both new instances of `C` and `E` will receive the counter with the value 2, which means that there are two instances of step `C` with the same counter value: The instance created by iterating the outer loop and the instance created by iterating the inner loop. **They must have different counters, otherwise they will not be distinguished.** The problem derived from the naive counter generation rule, that we initially described.

The solution is to generate a new identifier whenever a loop control flow is traversed. The generated identifier must be unique in the context of the workflow instance that is being executed. Instead of a counter we name it a *token*. Given that tokens are generated unique, the situation described in this third case can no longer occur.

4.2.3 Extension of the Execution Algorithm

In this section we describe how we extend the core algorithm to support the requirements identified in the previous section.

We begin by adding one slot to every step instance to contain the *current token*. Its purpose is to allow the engine to distinguish two instances of the same step. We add one more slot for each step that is the target of a loop control flow, to contain the *loop exit token*.[6] This will be set when entering a loop. It will be used when leaving the loop to propagate the same token that was being used when the loop began.

[6]Note that the loop exit token is defined for each step, not for each step instance.

When the engine starts the execution of a new workflow instance it marks all loop exit tokens as undefined and it generates a token for the start step instance.[7]

During workflow execution, for each step instance executed, the engine must now decide the following:

- Whether to set the step's loop exit token.
- Which token to propagate to the next step instance.

The engine takes the first decision when starting the execution of a step instance. If the step is the target of a loop control flow, then the engine stores the step instance's current token in the step's loop exit token (unless the loop exit token is already set, which means that we are not entering the loop for the first time but instead we are performing one more iteration). This procedure effectively sets the loop exit token when starting the first iteration of a loop.

The second decision is taken after the step instance finishes its execution and while iterating each outgoing control flow. If the control flow condition evaluates to `true`, (which means that we are going to traverse the flow) then:

- If the outgoing control flow is a loop, then when we are going for another iteration, so we need to use a different token. The engine generates a new token for the target step instance.
- If the outgoing control flow is not a loop, and there is **another** control flow that is a loop leaving this step, then it means that the control flow is leaving a loop area and, therefore, the engine will propagate the loop exit token that was set at the entry of the loop.
- In all other cases, the engine propagates the current token to the next step instance.

If a step is both the source and the target of loop control flows, then it is considered to be neither the entry nor the exit of a loop. The loop entry will be somewhere before, and the loop exit will be somewhere ahead, in the model[8]. In these cases the engine does not need to set the loop exit token at this step's beginning, nor does it propagate any loop exit token after this step's end.

Tables 4.1 and 4.2 summarise the behaviour we have described. Table 4.1 describes when to set the loop exit token, depending on whether the step is the source/target of any loop control flow. In fact, the only case in which the loop exit token is set is when starting a step that is the target of a loop control flow, but is not the source of one.

	is loop target?	**isn't loop target?**
is loop source?	–	–
isn't loop source?	set loop exit token (unless already set)	–

Table 4.1: Table for setting the loop exit token when entering a step.

[7] Whenever the engine generates a new token, it must always create a unique token, i.e., a token that has never occurred in the context of this workflow instance.

[8] Hence, we consider the longest possible loop body that can be iterated.

Table 4.2, shows which token is passed to the next step instance. We always pass the current token unless the step that has just finished is the source of a loop control flow, but not the target of one. In that case we either generate a new token if navigating an outgoing loop control flow or we propagate the loop exit token if navigating an outgoing non-loop control flow.

	is loop target	**isn't loop target**
is loop source	current token	loop flow => generate new token; otherwise => propagate loop exit token
isn't loop source	current token	current token

Table 4.2: Table for token propagation when leaving a step.

Computation of the Loop Exit Token

We have mentioned the propagation of the loop exit token, but we haven't defined how it is obtained, i.e. when leaving a cyclic area of the workflow model, how does the engine discover which value to propagate. The computation of the loop exit token is a function that receives as an argument a step that has an outgoing loop control flow. The goal is to find the token that was stored when entering that cyclic area.

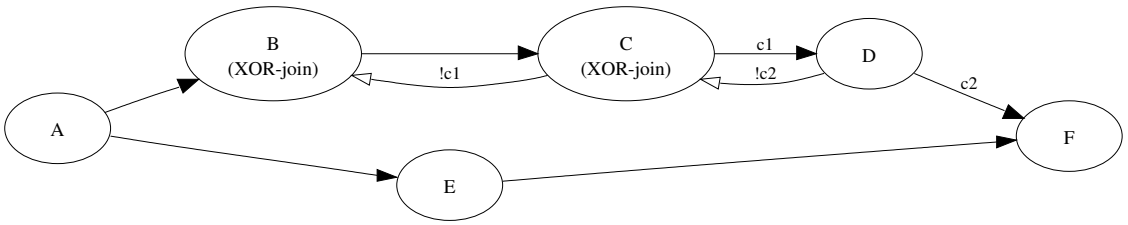

Figure 4.7: Computation the loop exit token.

Consider the example in Figure 4.7. We want to execute step `F` when `E` is finished, `D` is finished, and `c2 == true`. Remember that the synchronisation rules mentioned in the main algorithm apply only when joining paths with the same token. Hence, for the AND-join synchronisation rule at `F` to work as we intend, the control flows `D -> F` and `E -> F` must both target a step instance `F` with the same token. Thus, when the function computes the loop exit token to propagate from `D` to `F`, it must return the loop exit token that was stored in `B`.

We define the computation of the loop exit token for a given step `X` as: Start at `X` and recursively navigate each outgoing loop control flow to its target step until a step `V` that does not have an outgoing loop control flow is reached (this is the beginning of that cyclic area in the workflow model). Return the value of the loop exit token stored for `V` and also set the value of `V`'s loop exit token as undefined (this effectively serves as a reset in case this loop is nested inside a another loop). If `V`'s loop exit token was already undefined (which can happen for unstructured loops), then return `X`'s current token as the computed loop exit token (i.e., if by some arbitrary cycle structure we have already left the cyclic area through another path, then the engine cannot find any token to propagate, in which case it propagates the current token).

It derives from the previous definition that if there is a cyclic sequence of loop control flows,

then the computation of the loop exit token is not possible. Therefore, such situations should never be modelled. Additionally, there can at most one incoming and one outgoing loop control flow for each step. These are the only restrictions imposed at the back-end language and they can easily be worked-around by adding more steps to the workflow model.

4.2.4 Examples

We finalise the presentation of the execution module with some examples involving loops, to demonstrate the application of the algorithm with regard to the tokens.

Example 1

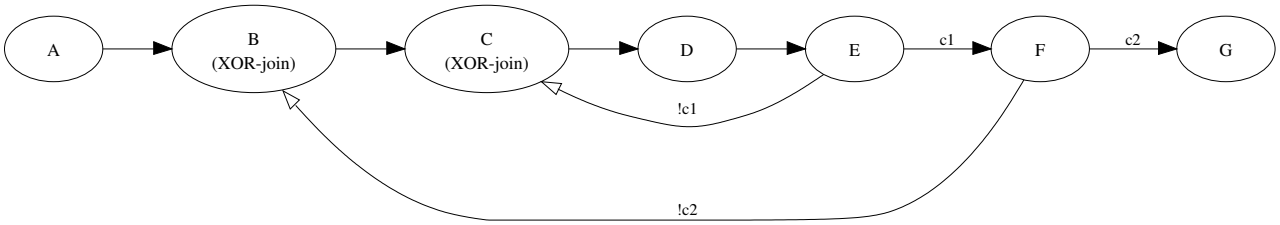

Figure 4.8: Example 1: Workflow model for nested loops.

Consider the example shown in Figure 4.8. It shows two structured loops, one nested inside the other. In this simple workflow it is clear that steps `E` and `F` are the exit steps of the two cyclic parts of the model. Table 4.3 shows the values of the tokens[9] associated with each step instance that is created when the executed sequence is `A, B, C, D, E, C, D, E, F, B, C, D, E, F, G`. Time is represented from top to bottom. Each new line is an instant that passes from the scheduler's point of view. On the left side are identified the loop exit tokens defined at any given instant. The remaining columns identify the value of the current token for the executed step instance.

Starting from the top, step `A` is started with a unique token. After step `A` ends, the instantiation of `B` is performed with the same token. Given that `B` is the target of a loop control flow and is not the source of a loop control flow, then its current token is stored as the loop exit token associated with step `B`. The same occurs when executing the first instance of `C`. All step instances up until `E` are started with the same token.

When step `E` finishes and `!c1 == true` then, because `E -> C` is a loop control flow, a new token is created for `C`. Given that the loop exit token for step `C` is already defined, then it is not defined again. Later, when `E` finishes and `c1 == true` then, because `E -> F` is not a loop control flow, there is a loop control flow whose source is `E`, and there is no loop control flow whose target is `E`, the engine computes the loop exit token (it comes from step `C`) and uses it as the token for `F`. This transition resets the loop exit token for `C` (i.e. it consumes it).

After `F` finishes, we go back to `B`. `B`'s loop exit token is not redefined, but `C`'s is, because it is not set. In result to that, when the flow from `E` to `F` is taken, `F` is instantiated with token `3`. Finally, step `G` is instantiated with the loop exit token stored at `B`, i.e., `1`.

[9] We use numbers to represent the tokens.

	A	B	C	D	E	F	G
	1						
B=1		1					
B=1,C=1			1				
B=1,C=1				1			
B=1,C=1					1		
B=1,C=1			2				
B=1,C=1				2			
B=1,C=1					2		
B=1						1	
B=1		3					
B=1,C=3			3				
B=1,C=3				3			
B=1,C=3					3		
B=1						3	
							1

Table 4.3: Example 1: Table of tokens for each instant.

Flow leaves the loops with the same token as when it entered them.

Example 2

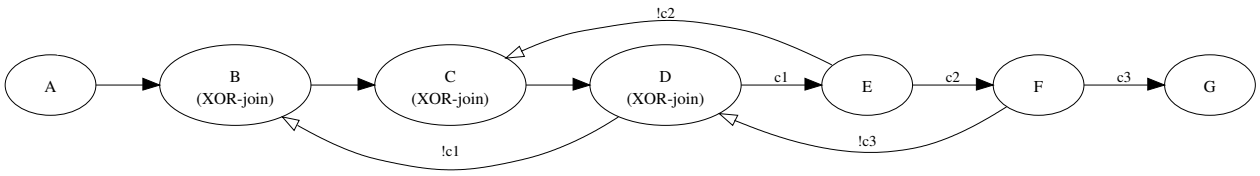

Figure 4.9: Example 2: Workflow model for arbitrary loops.

The next example shown in Figure 4.9 is more complex, because its loops are not structured. There are three interconnected loops: `BCD`, `CDE`, and `DEF`.

Table 4.4 contains the tokens for each instance when the executed sequence of steps is `A, B, C, D, B, C, D, E, C, D, B, C, D, E, F, D, B, C, D, E, F, G`. The table demonstrates that, as intended, `G`'s token is the same as `A`'s. Even in the presence of unstructured loops the exit from the cyclic area of the model is performed with the correct token.

Note that if we model a loop with multiple exit points and concurrent work that starts from within the loop's body, then we produce a situation where the **first** exit point that is executed will carry the loop exit token, but all other exit points will carry another token. This is unavoidable if the model is not structured and has many exit points. Still, the semantics is clearly defined.

Example 3

Lastly, it is worth mentioning that, it is possible to model a cyclic workflow without using loop control flows. This statement may seem odd at first, because of everything that has been said about

	A	B	C	D	E	F	G
	1						
B=1		1					
B=1,C=1			1				
B=1,C=1				1			
B=1,C=1		2					
B=1,C=1			2				
B=1,C=1				2			
B=1,C=1					2		
B=1,C=1			3				
B=1,C=1				3			
B=1,C=1		4					
B=1,C=1			4				
B=1,C=1				4			
B=1,C=1					4		
B=1						1	
B=1				5			
B=1		6					
B=1,C=6			6				
B=1,C=6				6			
B=1,C=6					6		
B=1						6	
							1

Table 4.4: Example 2: Table of tokens for each instant.

the need to identify loop control flows, but in some circumstances, depending on what we want to model, it might make sense to omit loop control flows.

Consider the following situation: There are three tasks, `A`, `B`, and `C`, which need to be executed sequentially and repeatedly. At any given point the repetitive execution of these steps can be "suspended" in favour of another step `D`. After `D` is finished, execution of the iteration resumes at some point depending on some condition. Figure 4.10 shows the described model. In this example step `D` outputs a message with the step to resume in data item r, which is used to choose the resume point. Condition s represents whether we should suspend to `D`. Note that the steps need to have OR-join synchronisation rules; otherwise, execution would deadlock.

There are no loop exit tokens kept, because there aren't any loop control flows. All current tokens have the same value (i.e. the value of the initially generated token).

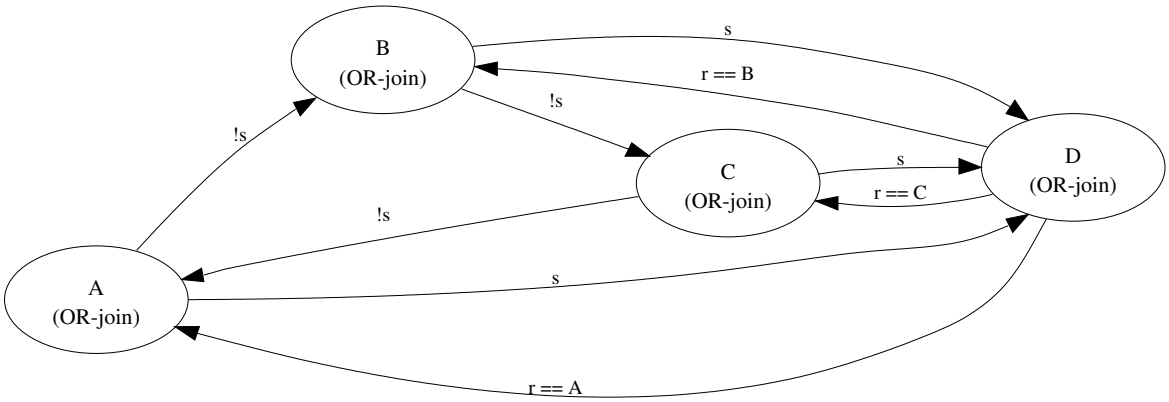

Figure 4.10: Example 3: Cyclic workflow model without loop control flows.

5 Implementation

This chapter describes the implementation of a WfMS that was developed using the WfVM approach proposed in this thesis. In the following sections we describe the system with emphasis on the following elements: The definition and execution modules, and the process of compiling workflows defined in the front-end language to the back-end language.

5.1 WorkSCo

Workflow with Separation of Concerns (WorkSCo) [Wor] is both an object-oriented framework and a WfMS. The core of the framework implements the WfVM, supporting the execution of workflows defined in the back-end language that we described in the previous chapter. The WfMS is built on top of the framework. Besides providing an API for typical workflow operations, it materialises the front-end definition module with a concrete front-end language.

The framework was developed with modularity in mind. Only the core aspects of workflow management are implemented at the kernel, which provides common workflow concepts to programmers. Extended functionality (such as monitoring, access control, worklists, and dynamic modification of the workflows) is implemented as separate modules. These modules can be composed as needed, thus making the framework lighter and easier to extend. We implemented the framework using an extension of the Java language with support for *Feature Oriented Programming* (FOP) [BSR03] concepts. The desired *layers* implementing the several workflow modules are composed together using the AHEAD Tool Suite [Bat, Bat04]. Some workflow modules—most of them—were developed across several layers, with each layer implementing some *feature*.

In the context of the two European projects [COM, ACE] in which this WfMS has been used, the target users were software programmers. Therefore, it seemed appropriate to define a structured front-end language. We have chosen to base the front-end language in the work of Manolescu, which, in his PhD thesis [Man01], defined a structured workflow language intended for software programmers.

5.2 Definition Module

WorkSCo's definition module implements both the back-end language of the WfVM and the front-end language we already mentioned. It provides operations related to the creation of workflow definitions at both levels, plus the compilation mechanism associated with transforming the front-end into the back-end.

5.2.1 Front-end Module

The implementation of the front-end module is split in two layers. One layer implements the abstraction of a "generic" hierarchic module for front-end workflow definitions. The concepts in that layer are sub-classed in another layer that contains our concrete front-end implementation. The justification for this separation is that several parts of the extension modules can be programmed for the generic front-end and they do not depend on the specific chosen front-end language.

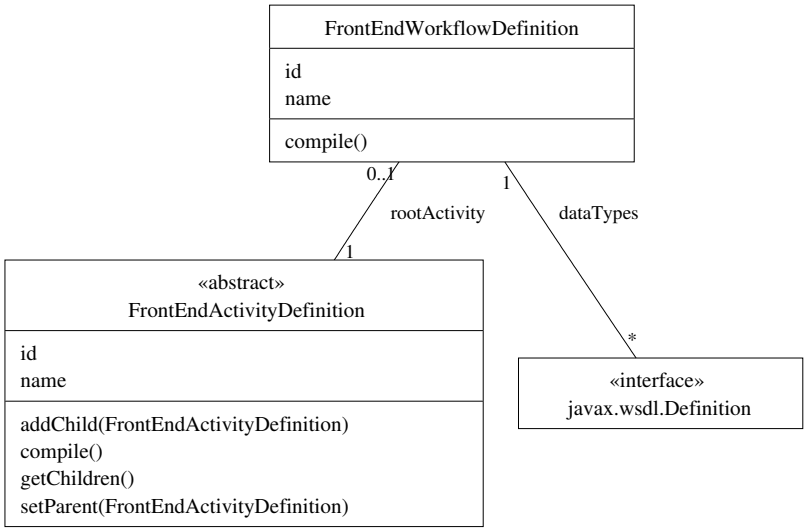

Figure 5.1: Class diagram of the "generic" front-end implementation.

The relevant classes for the "generic" front-end are represented in the class diagram in Figure 5.1. A `FrontEndWorkflowDefinition` has a unique `id` and a user-friendly `name` to help the user to identify it. The data types used in this workflow definition are represented as a WSDL `Definition` [CCMW01]. Each `FrontEndWorkflowDefinition` references the root activity of the hierarchical definition. This is an abstract class that needs to be sub-classed by a concrete front-end implementation. There are abstract methods that operate over the child elements of the definition.[1]

The `compile()` method in the `FrontEndWorkflowDefinition` generates a back-end `WorkflowDefinition` that has the same data types and the compiled version of the `FrontEndActivityDefinition`.

The class diagram for the concrete front-end is shown in Figure 5.2. The key abstraction of this particular front-end is the `Procedure`, which is a subclass of the `FrontEndActivityDefinition`. A tree of hierarchically composed procedures defines a workflow.

The hierarchy of procedures has control structures that are similar to those available in structured programming languages. The hierarchy is built using the Composite [GHJV95] design pattern to abstract the concrete type of the procedures that constitute the workflow definition, thus allowing the easy extension of the language with new procedures.

The `LeafProcedure`s model the domain-specific work. They are the leaves of the hierarchy. The `PrimitiveProcedure` is used for automatic execution of work, whereas the `WorklistProcedure`

[1] In the class diagrams of this chapter we only show the relevant methods, in order to make the explanation clearer.

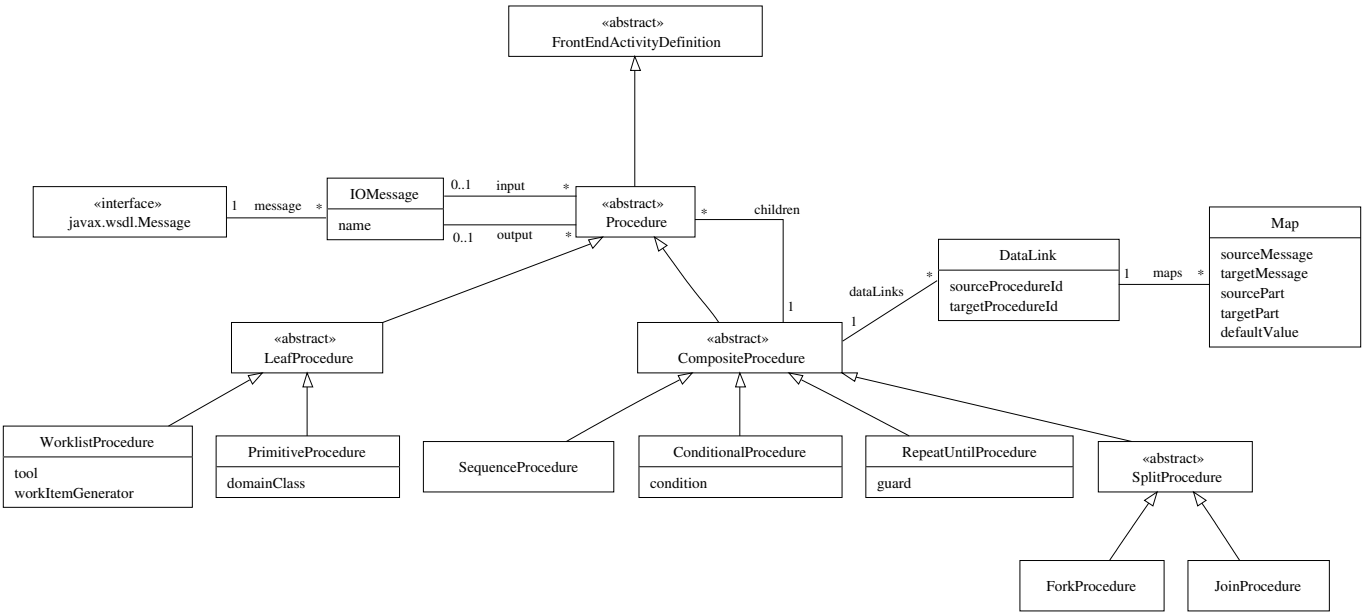

Figure 5.2: Class diagram of the concrete front-end implementation.

is used for work that needs human intervention.[2] A `PrimitiveProcedure` holds the name of a Java class. At runtime, when the `PrimitiveProcedure` is executed, it must instantiate the indicated class and execute it. The class indicated in the `domainClass` attribute must implement a specific interface (`IDomainObject`). The `WorklistProcedure`, when executed, will place a work item in a worklist and wait until it is performed. The `workItemGenerator` can be used to produce a specific work item. If not provided a `GenericWorkItem` is created. The optional `tool` attribute, is the name of an application that can be launched when the work item is being executed.

The other procedures are `CompositeProcedure`s and they model control flow primitives. We defined five types of concrete composite procedures:

- `SequenceProcedure`. It represents the execution of all its children in the sequential order in which they are stored.
- `ConditionalProcedure`. Enables the control flow to branch by having a `condition` that specifies one of two mutually exclusive paths to take.
- `RepeatUntilProcedure`. Holds one child `Procedure` (its body) and cyclically executes it until a given condition is achieved (i.e. while the `guard` is false).
- `ForkProcedure`. It spawns the parallel execution of all its children and waits for the first one to finish.
- `JoinProcedure`. Like the previous one, it spawns the parallel execution of all its children, but it waits for all of them to complete execution before completing itself.

[2]Currently, WorkSCo already supports other `LeafProcedure`s, such as the `WebServiceProcedure`. They are extensions to the core of the procedures' language that increase the modelling concepts available at the front-end. We decided to omit their presentation from this chapter, because the compilation mechanism is similar to the `PrimitiveProcedure` and, thus, including them would not add more useful information to understanding the front-end implementation.

Given the behavioural resemblance of the last two procedures, their common description is factorised in the `SplitProcedure` class.

Every procedure can consume and produce data. The input and output data is represented by an `IOMessage`, which is a wrapper for the WSDL `Message` interface. It adds a `name` to the message that can be used to refer to it.

The front-end language has an explicit data flow model. The `DataLink` class represents a flow of data from one procedure to another. There are restrictions to the data links. It is only possible to pass data either between siblings, or between a parent and its direct children procedures. The data link is always modelled at the level of the parent (even if the link is between two siblings). Thus, only the composite procedures can have data links. Note that the parent procedure can pass/receive data to/from any of its children, and that the data links between siblings are not restricted to any particular order. For each of the possibilities there is a default behaviour:

- **From a procedure to any of its children.** If no mappings are specified, then the default is to pass all the contents of the parent's input message to the child's input message. This is mainly used to initialise the child with the data that was passed to the composite.
- **Between siblings.** This can only occur in procedures that contain some ordering of their children (e.g. it does not make sense to pass data between branches of the `ConditionalProcedure`). If there are no mappings defined, then, by default, data is passed from the source procedure's output message to the target procedure's input message.
- **From any procedure to its parent.** If no mappings are specified, then the default is to pass all the contents of the child's output message to the parent's output message. This is usually used to add data produced by a child to the composite's output.

Figure 5.3 shows an example of the three data link types with their default behaviour. In this case the `SequenceProcedure` initialises its first child with its own input message (DL_A) and data is passed between its children (DL_B). Finally, the sequence outputs whatever its last child step produces (DL_C).

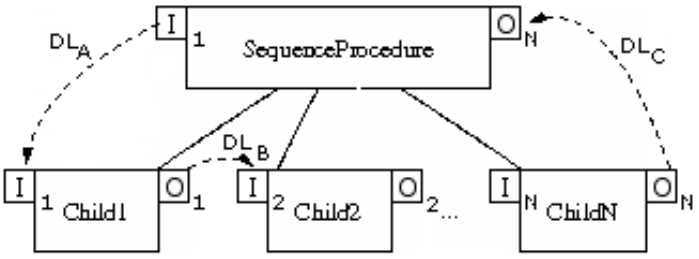

Figure 5.3: The data link types.

The `Map` can be used to override this behaviour. A `Map` for a given `DataLink` indicates which message of the source procedure should be read and which message of the target procedure should be written. Additionally, it is possible to copy only a part of the WSDL message using the `sourcePart` and `targetPart` attributes. The `defaultValue` serves two purposes. It can be used to: 1) define

a constant value to pass to a target message part when the source message part is not indicated; 2) define a default value to use if the source part of the source message is `null`.

5.2.2 Back-end Module

The back-end module implements the meta-model described in Section 4.1. Figure 5.4 shows the class diagram for the back-end language. Given that the back-end concepts have already been described as part of the WfVM, we will only present the relevant implementation-specific details.

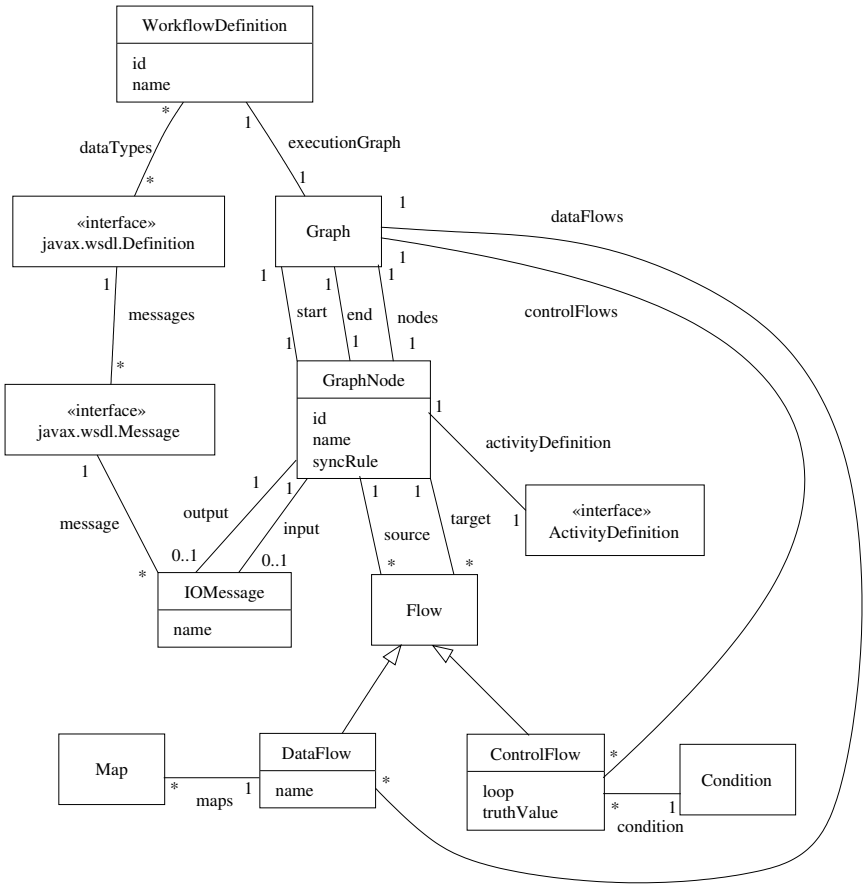

Figure 5.4: Class diagram of the back-end language implementation.

The root element is a `WorkflowDefinition`. It is analogous to the `FrontEndWorkflowDefinition`, except that instead of a structured definition it contains a `Graph`, which holds the entire workflow structure apart from the definition of the `dataTypes`.

The `GraphNode` represents a workflow step from the back-end meta-model. Each `Graph` has a `start` and an `end`, which must be `GraphNode`s from the set of `nodes`.

The `syncRule` attribute indicates the step's synchronisation rule (the default is the AND-join). `GraphNode`s are connected through `Flow`s, which explicitly model either a `DataFlow` or a `ControlFlow`. In the case of the `DataFlow`, the `Map` has the same function as for the front-end language.[3] As for the `ControlFlow`, it has two boolean attributes: The `loop` represents whether it is a loop control

[3]With the exception that there are no data flow restrictions and whenever there are no `Map`s the default is always to pass the data from the source node's output message to the target node's input message.

flow and the `truthValue` states the expected outcome of evaluating the transition `Condition`, i.e., whether the condition must evaluate to `true` or `false` for the control flow to be navigated.

Each `GraphNode` has an `input` and an `output IOMessage`. As before, this is a wrapper for the WSDL `Message` interface.

5.2.3 Front-end/Back-end Language Mapping

This section describes the rules implemented for compiling the "procedures-based" structured front-end workflow definition into the back-end definition.

The method `compile()` in the `FrontEndWorkflowDefinition` class is the starting point for the transformation. This method is invoked by the system whenever a front-end workflow definition is loaded. Its output is a `WorkflowDefinition`. The following describes the implementation of the `compile()` method. As can be seen in Figure 5.5, at the level of the `FrontEndWorkflowDefinition` the compilation consists in returning a new instance of `WorkflowDefinition` after compiling the `rootActivity`. The `id` and the `name` attributes are the same. The `dataTypes` have a one-to-one mapping because the front-end and the back-end use the same representation (WSDL).[4]

```
public WorkflowDefinition compile() {
   return new WorkflowDefinition(_id, _name)
     .setExecutionGraph(_rootActivity.compile())
     .setDataTypes(_dataTypes);
}
```

Figure 5.5: Compilation of the `FrontEndWorkflowDefinition`.

All the `Procedure`s are `FrontEndActivityDefinition`s and therefore they all implement the `FrontEndActivityDefinition`'s **abstract** method `compile()`. The compilation starts in the root of the tree of procedures (the `rootActivity`) and is defined recursively: Each composite procedure compiles its children (which generates part of the back-end `Graph`), and then links some of the resulting nodes, depending on the procedure's semantics. The recursion's base is the compilation of a `LeafProcedure`. When the recursion ends, the `rootActivity` has been transformed into the `executionGraph`. Next, we describe the concrete compilation for each type of procedure.

Leaf Procedures

`PrimitiveProcedure`s and `WorklistProcedure`s have the most direct and simple compilation rules. The resulting `Graph` for the compilation of any of them contains only one `GraphNode` (which is both the start and the end nodes of the resulting `Graph`). The node's input and output messages are the same as those defined for the procedure's input and output messages, respectively. The only difference resides in the domain-specific work. The node compiled from a

[4] Note that only the data types are the same, and that the data flows must be transformed, which will be done inside the compilation of the `rootActivity`.

`PrimitiveProcedure` will contain a subclass of `ActivityDefinition` that when called will invoke a domain-specific application (`PrimitiveActivityDefinition`), whereas the activity definition for the `WorklistProcedure` will create a work item, place it in the worklist and block until the work item is removed from the worklist via the engine's API (`WorklistActivityDefinition`).

Composite Procedures

The compilation of each composite procedure produces two `GraphNodes`, which contain between them the `Graphs` that correspond to the compilation of the children procedures. The `ActivityDefinition` of the start and end nodes of the composites is the `IdentityActivityDefinition`, i.e. an `ActivityDefinition` that when executed, transfers all data from the input message to the output message. The input and output messages of the start node are of the same type as the composite procedure's input message. The input and output messages of the end node are of the same type as the composite procedure's output message. Unless mentioned otherwise, the control flows are generated with `true` transitions, so that when control reaches the transition, its condition always evaluates to `true`, meaning that the transition should be taken to the target node.

In case there are data links then, by default, this is what happens (assuming that the children procedures are already compiled into `Graph`s and that the start and end nodes of the procedure currently being compiled are already created):

- **If the data link is from the composite procedure to a child**, then a data flow is generated from the output of the composite's start node to the input of the child's start node.
- **If the data link is from one child to another**, then a data flow is generated from the output of the source child's end node to the input of the target child's start node.
- **If the data link is from a child to the composite procedure**, then a data flow is generated from the output of the child's end node to the input of the composite's end node.

`SequenceProcedure` (Figure 5.6). The compilation for the control flow part consists in:

- Link its start node to the start node of the graph corresponding to its first child step.
- Link the end node of the graph corresponding to its last child step to its end node.
- Link all the child steps between them in the order imposed by their sequential declaration.

`ConditionalProcedure`(Figure 5.7). This procedure has only two branches: One to execute if the condition evaluates to `true` and another to execute if the condition evaluates to `false`. Hence, the following control flows are always generated:

- One from the `ConditionalProcedure`'s start node[5] to each of the start nodes of the child branches.

[5]Of course, that procedures do not have nodes. When we refer to a "procedure's node" we obviously refer to a node of the "graph associated with that procedure's compilation". We just use the former expression because it is simpler.

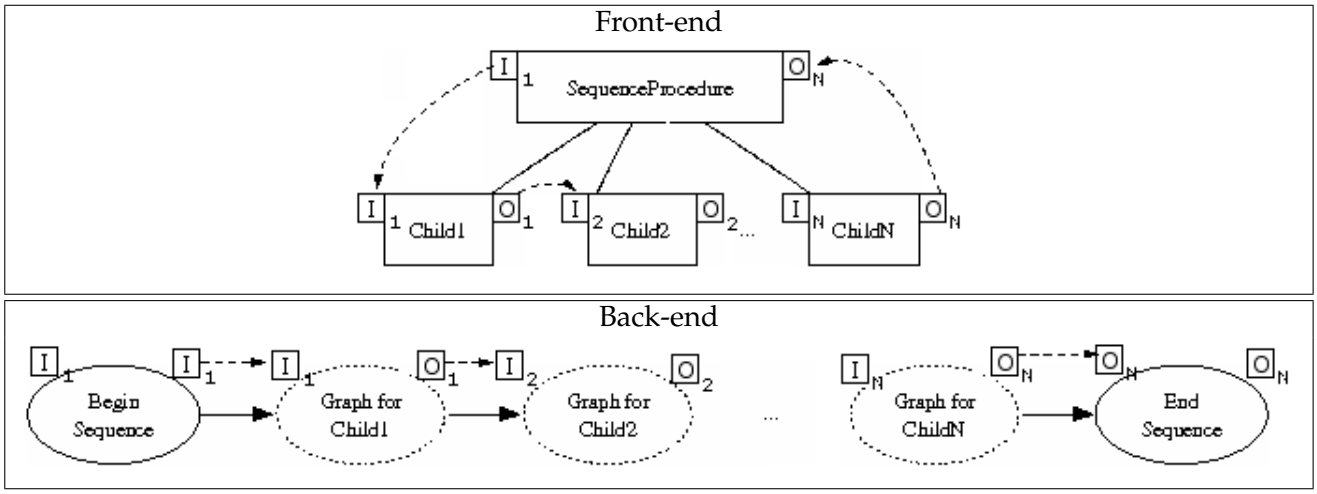

Figure 5.6: Compilation of the `SequenceProcedure`.

- One from each child branch's end node to the `ConditionalProcedure`'s end node.

Each of the two control flows that leave from the `ConditionalProcedure`'s start node must have a transition condition. For the "true" branch we use the `ConditionalProcedure`'s condition with the `truthValue` set to `true`. For the "false" branch we use the `ConditionalProcedure`'s condition with the `truthValue` set to `false`. Since both conditions are mutually exclusive, only one path will be taken, and for that reason the `ConditionalProcedure`'s end node must be an XOR-join node. The default AND-join would block execution forever waiting for the other branch (that is never executed) to complete.

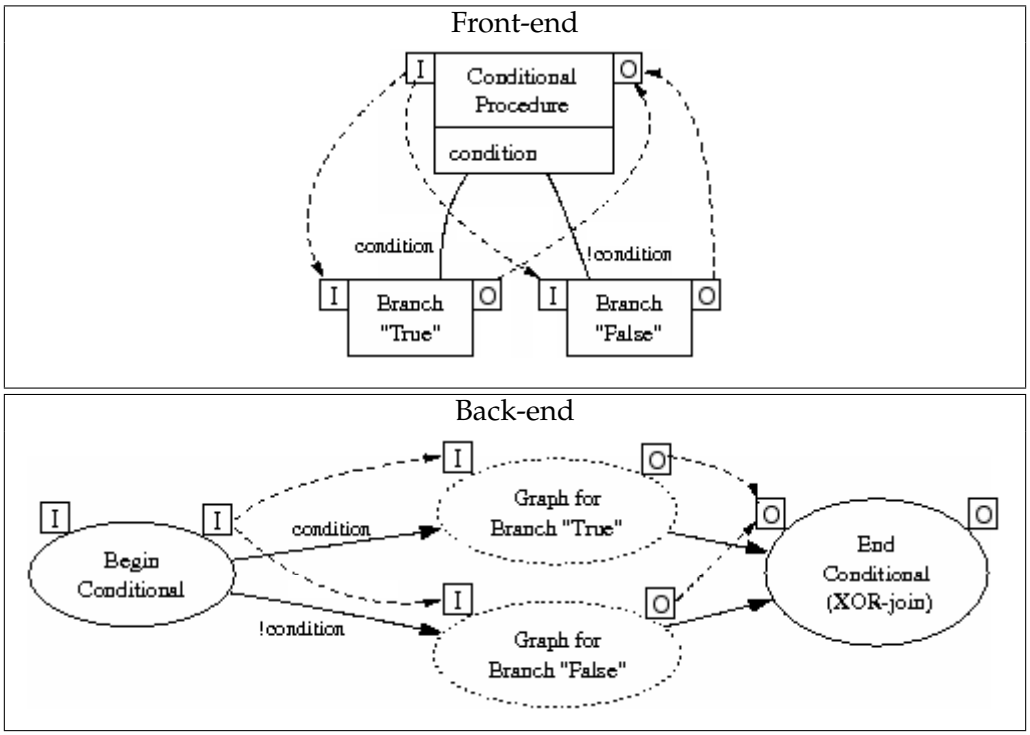

Figure 5.7: Compilation of the `ConditionalProcedure`.

`ForkProcedure`(Figure 5.8). When the `ForkProcedure` begins it must put all of its branches in parallel execution; therefore the generation is as follows:

- One control flow is generated from the `ForkProcedure`'s start node to each of the start nodes of the child branches.
- One control flow is generated from each child branch's end node to the `ForkProcedure`'s end node.

The `ForkProcedure` ends as soon as one of its branches finishes executing. For that reason the `ForkProcedure`'s end node has an XOR-join synchronisation rule.

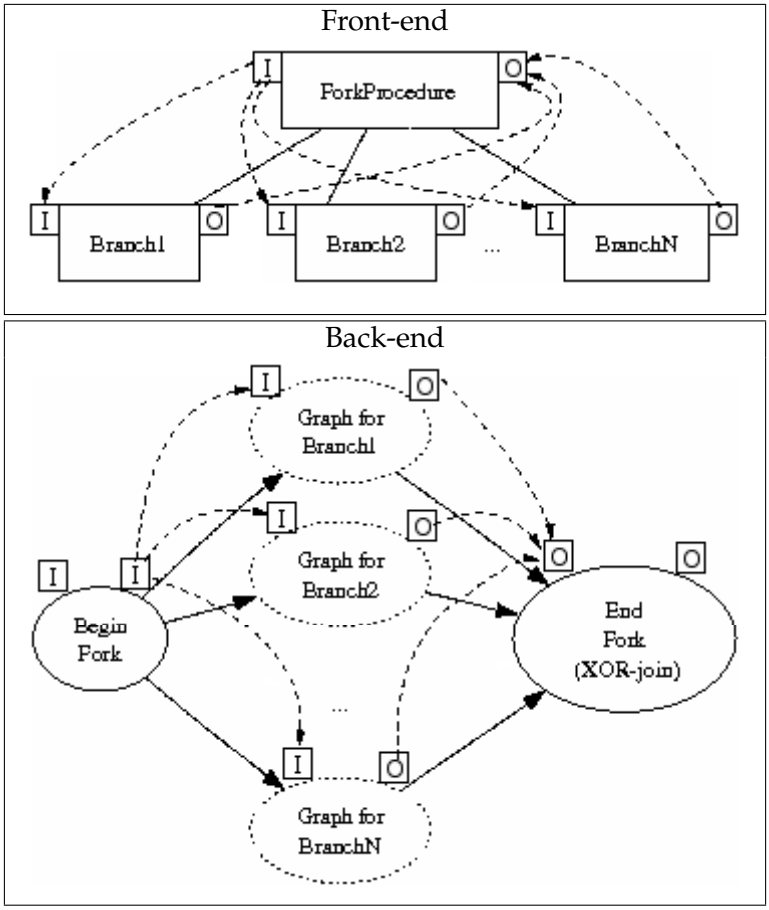

Figure 5.8: Compilation of the `ForkProcedure`.

`JoinProcedure`(Figure 5.9). This procedure's behaviour is identical to the `ForkProcedures`'s, except that it waits for all of its branches to finish executing before proceeding. The only change is, thus, in the end node generation, which has the default AND-join synchronisation rule.

`RepeatUntilProcedure`(Figure 5.10). The compilation of this procedure involves creating two more nodes to deal with the actions of entering and exiting the body of the loop, the begin body and end body nodes respectively. Also, two data flows are always generated:

- One from the `RepeatUntilProcedure`'s start node to the begin body node.

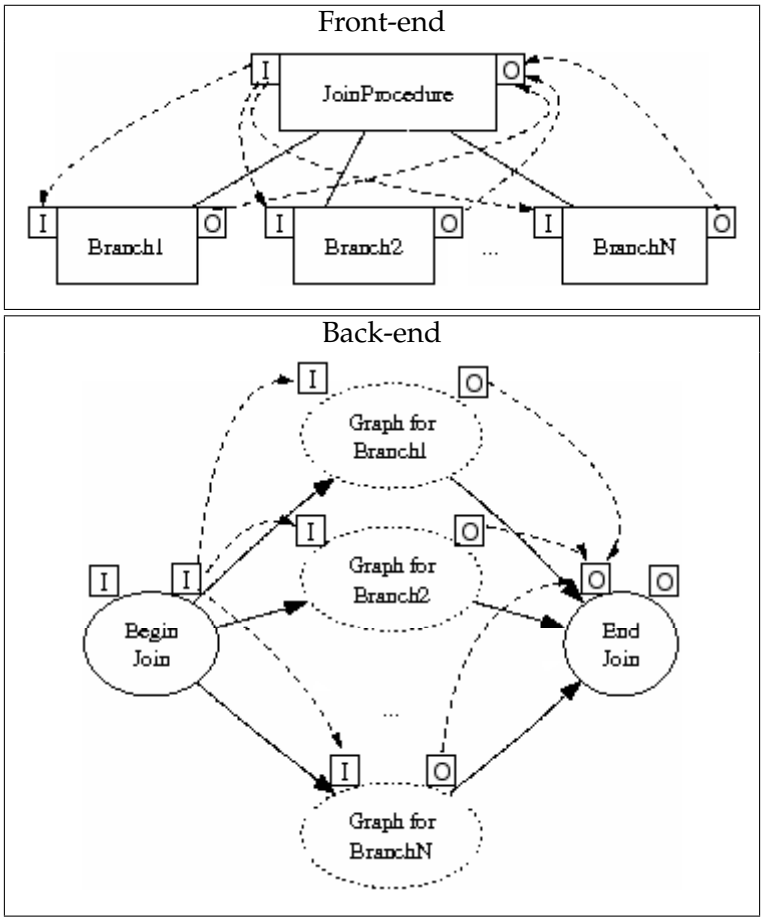

Figure 5.9: Compilation of the `JoinProcedure`.

- One from the end body node to the `RepeatUntilProcedure`'s end node.

These two data flows guarantee that there is no loss of data that goes to/comes from the `RepeatUntilProcedure`. The data links defined in the `RepeatUntilProcedure` are compiled according to the general rules that were described in the beginning of this section, but using the begin body and end body nodes instead of the start and end nodes, respectively. The generation of control flows for this procedure is as follows:

- One from the start node to the begin body node.
- One from the begin body node to the body procedure's start node.
- One from the body procedure's end node to the end body node.
- One from the end body node to the end node with the loop condition set as the transition condition.
- One from the end body node to the begin body node with the negation of the loop condition set as the transition condition.

Note that in the example provided in Figure 5.10 the body procedure has a data link from its output message to its input message. This represents a "feed-back" loop, i.e. a loop whose output

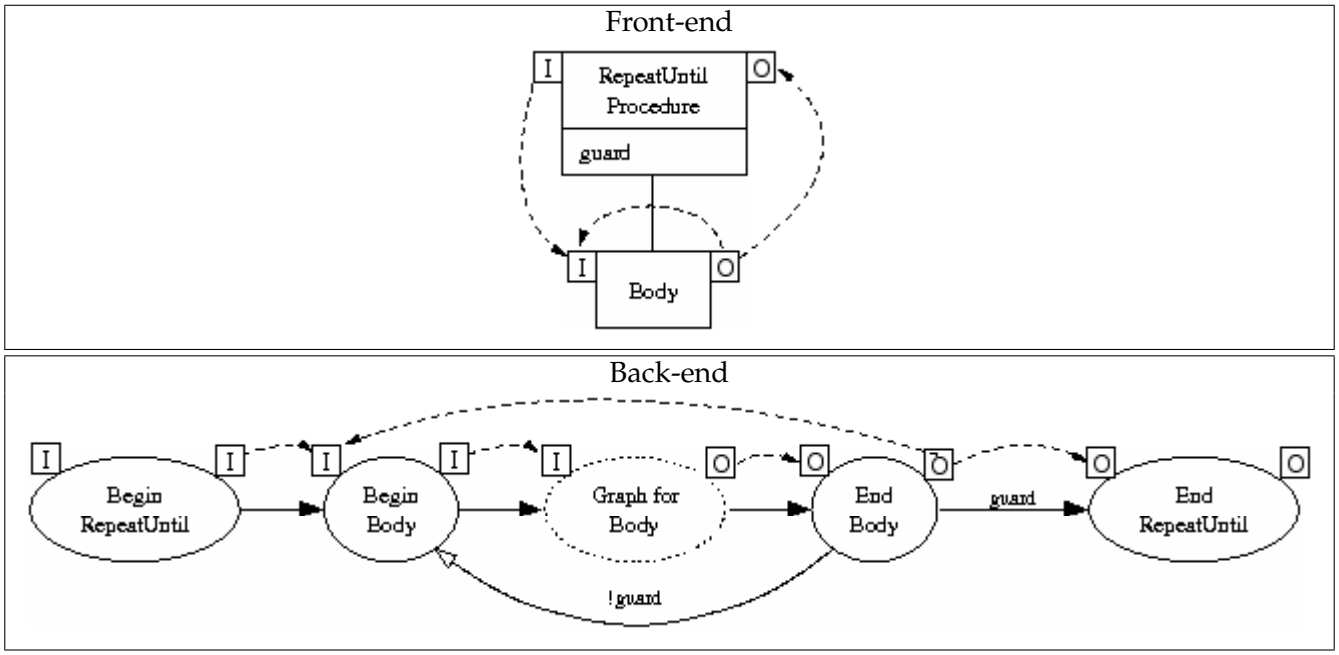

Figure 5.10: Compilation of the `RepeatUntilProcedure`.

data from an iteration is used as the input data for the following iteration. The compilation produces the equivalent data flow from the output of the end body node to the input of the begin body node. The begin body node has two data flows targeting its input message, but they do not conflict because at any given execution of that node only one of the data flows has data to provide: In the first iteration the data comes from outside the loop; in the following iterations the data comes from the loop.

5.3 Execution Module

WorkSCo's main class is the `WorkflowManager`. This is the core class and its most important operations are:

- Loading workflow definitions;
- Creating workflow instances;
- Starting the execution of workflow instances.

As it was mentioned in the previous section, whenever a `FrontEndWorkflowDefinition` is loaded, the `WorkflowManager` invokes the `compile()` method to obtain the back-end `WorkflowDefinition`. Before the enactment of a `WorkflowDefinition` can take place, an instance of that definition must be created. For that, the engine creates an instance of the `Executor` class, which is the object responsible for holding all the information relevant for executing a single workflow instance, as well as carrying out the graph navigation algorithm according to the specification in Section 4.2. There is one `Executor` for each workflow instance. The `Executor` also stores the workflow data that must be passed from one node to another.

When a step (represented as a `GraphNode`) is scheduled for execution, the `Executor` creates and `ExecutionElement`. The `ExecutionElement` represents a unique step instance within the workflow instance. Each `ExecutionElement` holds an `ExecutionContext`, which has the data received and produced by the step instance. A workflow instance can have concurrent paths of execution, which means that, at any given instant, it can have several `ExecutionElement`s. Therefore, the `Executor` manages the graph navigation by keeping a set of `ExecutionElement`s. This set corresponds to all the currently executing step instances.

The concurrency management, whether it is within the same workflow instance or between workflow instances, is facilitated by the application of Doug Lea's concurrency patterns [Lea99] using worker threads with queueless synchronous channels to hand off work to the threads. Each thread executes one workflow step instance (one `ExecutionElement`), and once it finishes executing one step it is recycled to be used on another.

The `Executor`'s states are presented in Figure 5.11.

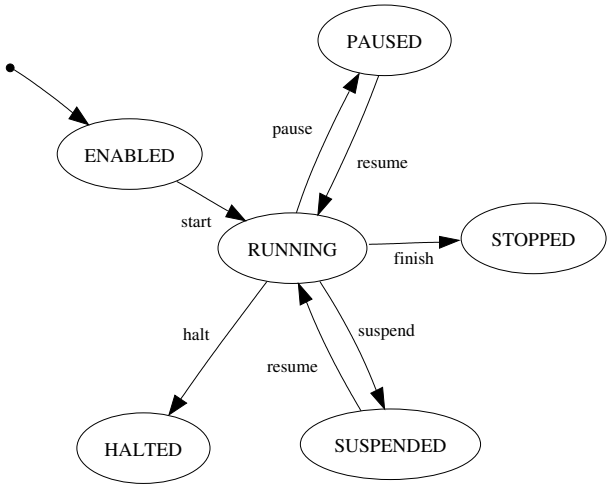

Figure 5.11: States of a workflow instance (`Executor`).

When a workflow instance is created it is set to the state `ENABLED`. This state represents an instance that is ready to begin its execution, but hasn't started yet. When the instance is started it changes to the state `RUNNING`. During its execution an instance can move temporarily to the states `PAUSED` or `SUSPENDED`. The difference between these two states is that in the `PAUSED` state the `Executor` will not advance workflow execution, but there may be some nodes still executing, whereas in the `SUSPENDED` state there are no nodes executing, i.e., entering the `PAUSED` state is immediate after it is requested, whereas entering the `SUSPENDED` state might take an arbitrary amount of time (because it depends on how long the executing nodes take to finish) and thus the caller is blocked until the state can be achieved. Initially, the `PAUSED` state did not exist. It was created, because some extension modules can take advantage of it. For example, the evolution module can make dynamic changes to the workflow, as long as it can guarantee that the workflow will not advance, even though it is still executing some domain activities.

When a workflow instance ends, it moves to the `STOPPED` state. If any error occurs during the execution, from which the engine cannot recover, then it sets the state to `HALTED`.

6 Evaluation

This chapter provides a workflow patterns-based evaluation of the expressiveness of the back-end language. For the proposed WfVM to be useful the most important thing is to have a back-end language onto which many front-end languages can be mapped. Another way of putting this is to say that the back-end language needs to be expressive enough to be able to represent typical workflow constructions. The research on workflow patterns [AHKB03] is the most exhaustive analysis of typical workflow constructions that we know of. In this chapter we evaluate the WfVM in light of those workflow patterns. For each workflow pattern a brief description is given and we discuss how (if) it can be supported by our back-end language.

Some reminders before proceeding: a) whenever not mentioned the control flow condition is the default condition `true`; b) the flow of *workflow control data* used in some patterns is not represented in the model, because all steps have access to it.[1] Whenever it is relevant there is text explaining how this data is used; c) "normal" control flows are represented using solid arrows and loop control flows are represented using hollow arrows; d) the default synchronisation rule is the AND-join.

6.1 Sequence

Description The Sequence pattern describes a situation in which an activity can only be enabled after the completion of another activity in the same workflow instance.

Solution The description of the pattern implies two steps in the workflow model (one for each activity). The dependence is expressed by directly connecting the first step to the second. After the termination of the first step the second is enabled.

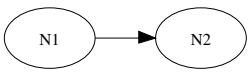

Figure 6.1: Sequence.

6.2 Parallel Split

Description A point in the workflow where a single path of execution is split into multiple concurrent paths. The execution order of each path relative to the others is not defined and multiple executions can present different behaviour.

[1] This is the opposite of the *workflow application data* which is explicitly passed from one step to another.

Solution One step in the graph must have multiple outgoing control flows. Whenever the step finishes, the execution of all the following steps will be enabled concurrently. Assuming there are already m paths of execution ($m > 0$) and n is the number of concurrent paths being created ($n > 0$), then after this point there will be $m - 1 + n$ execution paths.

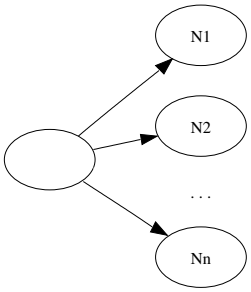

Figure 6.2: Parallel Split.

6.3 Synchronisation

Description A point in the workflow where multiple concurrent paths of execution are joined into one single path. In this pattern it is assumed that each concurrent path to be synchronised is executed only once.

Solution The construction is similar to the one in the previous pattern but with the control flows in reverse orientation. The default behaviour of the back-end language is to wait for all incoming control flows to be executed and only then to enable execution of the synchronising step, which is exactly what the pattern describes. Assuming there are already m paths of execution ($m > 0$) and we wish to synchronise n concurrent paths ($n \leq m$), then after synchronisation there will be $m - n + 1$ execution paths.

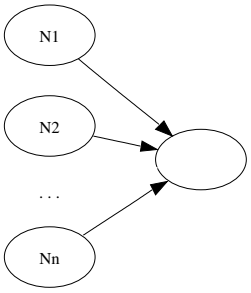

Figure 6.3: Synchronisation.

6.4 Exclusive Choice

Description A point in the workflow where only one of several paths is taken. The choice can be based on workflow control data or on workflow application data.

Solution Structurally this pattern is similar to the Parallel Split pattern (Section 6.2). There is one step with several outgoing control flows; one control flow for each decision to be taken. Decisions are represented as conditions in the control flows[2]. The conditions need to be mutually exclusive to ensure that at most one path is taken, as the pattern requires. Also, the conditions need to cover the scope of possible values to guarantee that at least one path is taken (e.g. two control flows, respectively with conditions $x > 0$ and $x \leq 0$). Finally, note that the data available to the conditions is the data that is produced in the output message of the step with the multiple outgoing control flows. Therefore, either that step's execution produces the data to be evaluated or there should be some data flow(s) targeting that step.

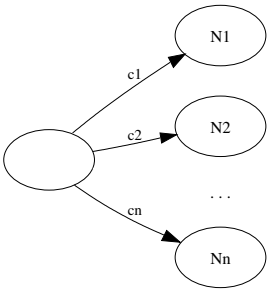

Figure 6.4: Exclusive Choice.

6.5 Simple Merge

Description A point in the workflow where two or more alternative branches come together without synchronisation. The pattern assumes that **only one of the branches is under execution**, which means that the merging point only has to deal with one execution path.

Solution As with the Synchronisation pattern (Section 6.3) it is necessary to model multiple control flows to the same step. The difference is that the default synchronisation behaviour (AND-join) cannot be used, because that would cause the workflow instance to deadlock given that the synchronising step would be waiting for all control flows to be navigated. So, the solution is simply to use an XOR-join, which will enable the step's execution immediately after the first control flow is received. Using an OR-join would have the same effect, because the pattern assumes that there is ever only one path to merge. See further patterns for other types of merging.

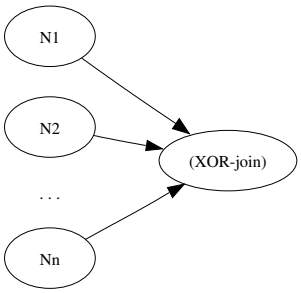

Figure 6.5: Simple Merge.

[2]Remember that a control flow is taken only if its condition evaluates to `true`.

6.6 Multi-choice

Description A point in the workflow where, based on a decision or workflow control data, some paths are chosen. The number of paths chosen depends on the result of evaluating the conditions.

Solution This is the generic case supported by the back-end language. Structurally, it is just like the Parallel Split (Section 6.2) and Exclusive Choice (Section 6.4) patterns. The difference is simply in the fact that any number of control flow conditions might evaluate to `true` and therefore the number of created paths of execution can be anything between 0 and n (n being the total number of alternatives). After this point the number of execution paths will be $m - 1 + t$ (m is the number of already existing paths and t is the number of control flows taken).

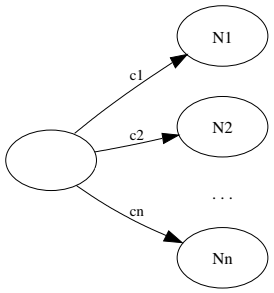

Figure 6.6: Multi-choice.

6.7 Synchronising Merge

Description A point in the workflow where multiple paths converge into one single path. The synchronisation point needs to wait for all the paths that were taken to finish before proceeding. Note that the number of paths taken might be less that those initially available, which means that the synchronisation cannot simply wait for all incoming control flows, because it would then dead-lock. The pattern assumes that a branch that has been already activated, is not activated again while the merge is still waiting for other branches to complete.

Solution The synchronisation step needs to know how many control flows it must wait for. This is directly dependent on the number of control flows taken in the split. However, in our back-end language there is no direct construction that establishes a relation between the number of split paths[3] and the number of joining paths[4]. So, the solution for this pattern requires a mixture of XOR-joins with AND-joins. The idea is to make sure that the number of paths to join is always constant, thus removing the need to wait for a variable number of control flows. Figure 6.7 shows the general form to achieve the desired result. It consists in the application of four of the already seen patterns. It starts with the Parallel Split (Section 6.2). For each of its paths it uses the Exclusive Choice (Section 6.4) and the Simple Merge (Section 6.5) patterns. The Exclusive Choice consists in a pair of outgoing control flows with the desired condition (c) and its negation ($!c$). The flow ends in a Synchronisation (Sec-

[3]Outgoing control flows taken.
[4]Incoming control flows expected.

tion 6.3). With this construction we achieve the desired behaviour. No matter how many "positive" conditions are taken the Synchronisation always waits for n control flows, because in the other cases the "negative" conditions are selected.

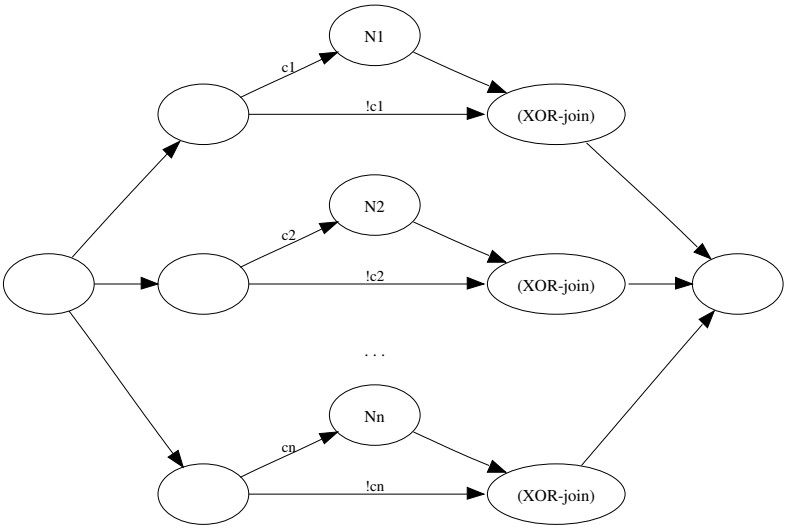

Figure 6.7: Synchronising Merge.

This pattern is not explicit about the expected behaviour when all the conditions $c_1, c_2, \ldots, c_n$ evaluate to `false`. In the implementation presented in Figure 6.7 it is clear that the workflow would proceed executing, because all the negated conditions would evaluate to `true`. If that is not the intended behaviour, then it is necessary to add a previous step with a control flow whose condition is that at least one of $c_1, c_2, \ldots, c_n$ evaluates to `true`.

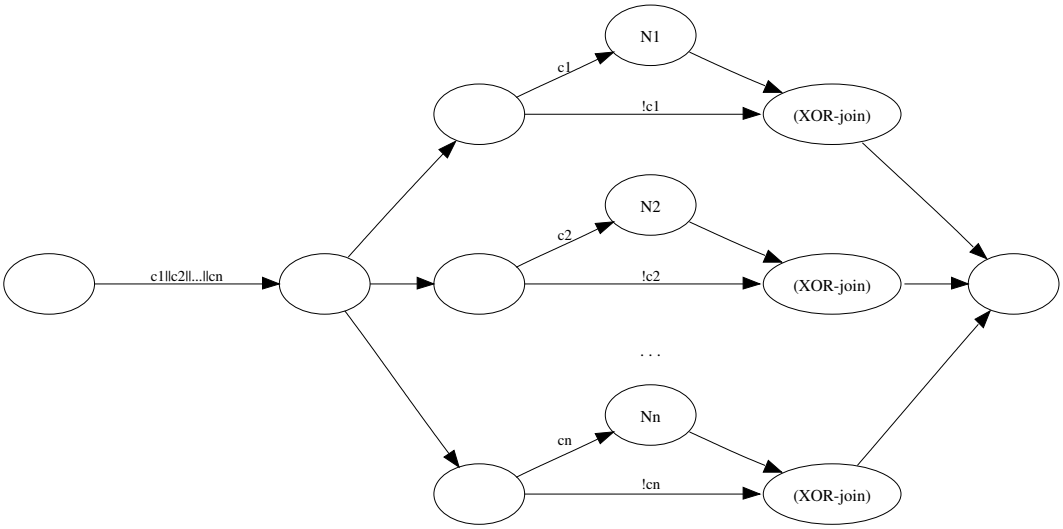

Figure 6.8: Synchronising Merge (checks if at least one "positive" condition is `true`).

The pattern's description simplifies by assuming that each path is activated at most once, i.e., each branch cannot be activated again while the merge is waiting for other branches. If that restriction is dropped, we can support multiple activations, as long as the step instances that are created have different tokens. Activations belonging to the same group (i.e. the ones with the same token) will be merged independently of the others. This typical situation can be produced when the Synchronising

Merge pattern is nested inside a loop. When the loop control flow is taken it produces a new token for the next iteration and, thus, the activations are not mixed up.

6.8 Multi-merge

Description A point in the workflow where two or more branches converge without synchronisation. This means that the activity following the merge should be activated for every incoming control flow that is taken.

Solution The implementation of this pattern is straightforward using the OR-join synchronisation rule. As soon as any incoming control flow occurs, the merging step is enabled for execution. If no incoming control flow occurs, then the merging step is never executed.

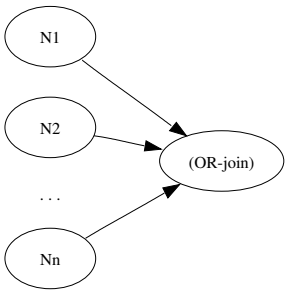

Figure 6.9: Multi-merge.

6.9 Discriminator

Description A point in the workflow that waits for one of the incoming branches to complete before activating the subsequent activity. From that moment on it waits for all remaining branches to complete and discards them. Once all incoming branches have been triggered, it resets itself so that it can be triggered again.

Solution In our back-end language, this pattern can be seen just as a generalisation of the Simple Merge pattern (Section 6.5). The XOR-join rule does exactly what this pattern describes. It already worked for the Simple Merge because the pattern was a special case, that assumed only one active branch. The merging step waits only for the first incoming control flow and discards the others. Workflow execution can proceed immediately after the first incoming control flow occurs.

This pattern requires, additionally, that the merging step resets itself after all incoming branches have been triggered. According to the author's description this requirement exists to support the repetition of the same behaviour, e.g., when the Discriminator is nested in an iteration.[5] In our language, this "reset" action is supported by the use of loop control flows. A loop control flow generates a new token, which eliminates any relation with previous executions when merging the incoming branches, thus "resetting" the behaviour.

[5] In many workflow systems, the Discriminator pattern will only work once, for each merging node.

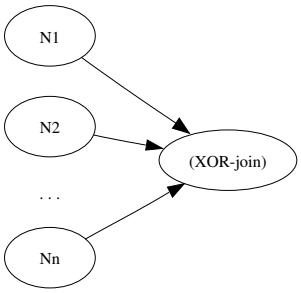

Figure 6.10: Discriminator.

6.10 Arbitrary Cycles

Description A point in the workflow where one or more activities can be done repeatedly.

Solution This pattern's name, Arbitrary Cycles, implies the possibility of iterating any group of activities regardless of their structure (e.g. regardless of other cycles), but its description is very vague. Let's consider two types of cycles. *Block-structured cycles* have only one entry point and one exit point. Other cycles may be nested inside such cycles, but they can not cross the boundary imposed by the external cycle. *Unstructured cycles* are basically backward jumps to any part of the workflow model creating a cyclic graph, regardless of block structures. The former represents typical iteration instructions in programming languages such as WHILE and REPEAT, whereas the latter represents jump instructions such as GOTO. The semantics of the first type is generally easier to grasp when looking at a model. However, when creating a model, sometimes the modellers tend to think in terms of activity dependencies, which is more biased towards an unstructured representation. The back-end language supports both approaches in the same manner. Any step can connect to another step using a loop control flow. Figures 6.11 and 6.12 show structured and unstructured cycles, respectively.

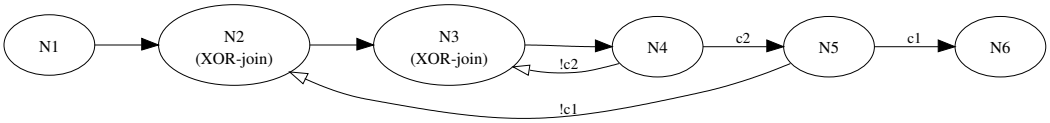

Figure 6.11: Arbitrary Cycles (structured).

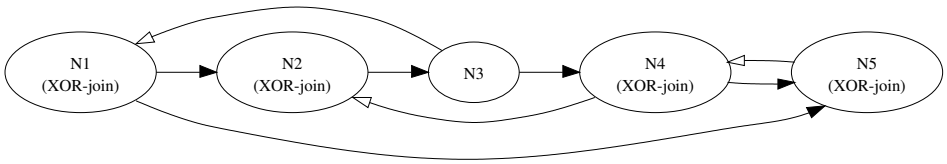

Figure 6.12: Arbitrary Cycles (unstructured).

The general rule to follow to describe a structured loop is to connect its steps in a closed path, using only one loop control flow. There should be only one entry and one exit from the loop, and the exit should be at the same step as the source of the loop control flow. No control flow can cross the boundary set by the closed cycle of control flows. In Figure 6.11 there are two such loops.

The unstructured model in Figure 6.12 shows that the back-end language can cope with arbitrary cycles in the model. However, it is a complex model and it is not easy to envisage a real workflow that would have such structure. There are no limitations imposed, when building unstructured loops, but care must be taken to ensure the desired semantics.

6.11 Implicit Termination

Description A given workflow should be terminated when there are no active activities, no other activity can be made active, and the workflow is not in a deadlock.

Solution This pattern is unlike others, because it does not necessarily reflect only a language feature; it also refers to the engine's characteristics. There are two possible workflow termination policies:

- **Implicit Termination** — the workflow instance is terminated when there is nothing else to do. A (weaker) variation is to also terminate a workflow instance when it can no longer proceed. In the latter case the workflow might end because it reaches a deadlock.
- **Explicit Termination** — the workflow instance is terminated when it explicitly reaches a certain step. This requires support from the modelling language. It must have a termination step.

Our back-end language has an explicit end step, and in WorkSCo we implemented the two alternative workflow termination policies in the following manner:

- **Implicit Termination** — whenever a step finishes its execution and produces no outgoing control flows, the engine tests whether the list of steps to be executed is empty.
- **Explicit Termination** — whenever a step finishes its execution, the engine tests whether it is the last step in the workflow.

In either of the policies if the test returns `true` then the engine sets the workflow instance's state to STOPPED. Our implicit termination policy does not support the Implicit Termination pattern, because it lacks the identification of deadlock situations. Our implementation would terminate a deadlocked workflow when using the implicit termination policy. The policy currently in use is the explicit termination policy only.[6]

6.12 Multiple Instances Without Synchronisation

Description Within the same workflow instance, multiple instances of an activity can be created, i.e., it is possible to spawn new paths of execution. Each of these paths is independent of the others. Moreover, there is no need to synchronise these paths.

[6]The authors of YAWL, which are the same that developed the study on workflow patterns, decided not to support the Implicit Termination pattern. They provided two reasons; first, to force the designer to think about termination properties of the workflow, and second, implicit termination can hide design errors because it is, in some cases, impossible to detect deadlocks.

Solution The back-end language does not have a construct to directly model this behaviour. Still, it can be easily achieved by modelling a cycle that spawns as many instances of the same activity as needed. The construction is very simple, because there is no need to wait for any of the instances to finish. The workflow will continue its execution as soon as all the activities have been spawned.

The model needs to represent the activity to spawn and some rule indicating whether to continue spawning. Figure 6.13 shows the generic structure. The activity `more?` models some domain-specific decision. Note that this function can be automatically generated when compiling the front-end to the back-end, because it generally has a pattern behaviour that depends on workflow application data, such as count up to a given limit. In this case e.g., the modeller would only have to specify in the front-end language the activity to spawn (`N1`) and the limit value. When no more spawning is required the execution proceeds to `N2`.

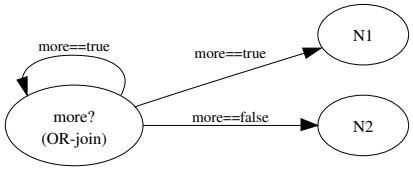

Figure 6.13: Multiple Instances Without Synchronisation (same token).

Note that there are no loop control flows in this model, which means that all executions of `N1` will have the same token. If for some modelling reason we want to make sure that the spawned paths will not interfere (e.g. because `N1` is, in fact, a complex structure), then we need to use a loop control flow as shown in Figure 6.14. The extra node is necessary, because if the loop control flow directly connected the steps `more?` and `N1`, it would interfere with the tokens used in the main flow.

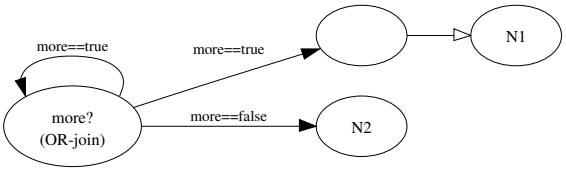

Figure 6.14: Multiple Instances Without Synchronisation (different token).

6.13 Multiple Instances With a Priori Design Time Knowledge

Description An activity is enabled multiple times in the same workflow instance. The number of instances of that activity is known at design time. When all instances of the activity are completed some other activity needs to be started.

Solution Figure 6.15 shows a trivial implementation. It is simply the composition of two patterns: The Parallel Split (Section 6.2) and the Synchronisation (Section 6.3). The activity to be enabled multiple times, `N1`, is duplicated for each path of execution. When all instances of `N1` finish, `N2` is started.

If this structure were modelled in a front-end language it would present a major drawback: The model would contain duplicated parts and it would be error-prone, because it would be easy to forget to change all the steps in case of a model update.

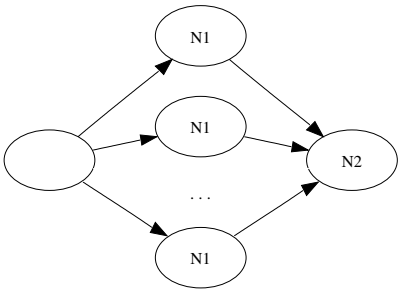

Figure 6.15: Multiple Instances With a Priori Design Time Knowledge.

This is not a problem for the back-end model because it results from a compilation of a front-end model. In the front-end language there should be a construct representing "multiple instances of one activity", which would not duplicate the information.

For a solution that does not use duplicate steps in the back-end, look at Figure 6.16. The solution presented there is also valid for this pattern simply by replacing the variable `TOTAL` with an hard-coded value (known at design time).

6.14 Multiple Instances With a Priori Runtime Knowledge

Description An activity is enabled multiple times in the same workflow instance. The number of instances of the given activity is only known at runtime and may vary from one workflow instance to another. However, the number of instances to create is known, during runtime, before the instances of the activity are created. When all instances of the activity are completed some other activity needs to be started.

Solution In this pattern the engine has to wait until several concurrently executing activities finish, before enabling the continuation of the workflow. The main difference to the previous pattern is that the trivial solution — to replicate the step containing the activity to be executed multiple times — is no longer possible because the number of concurrent activations is unknown during the design phase. This forces the model to contain only one step with the activity to be executed. The apparent difficulty is to know, each time one activity instance finishes, if the workflow can proceed. It is not possible to get that information from the structure of the model because there is only one step. The solution is for the workflow instance to keep track of how many activities have finished and provide a conditional control flow that can only take place when the last activity finishes. This is workflow control data that is not manipulated by the domain-specific work.

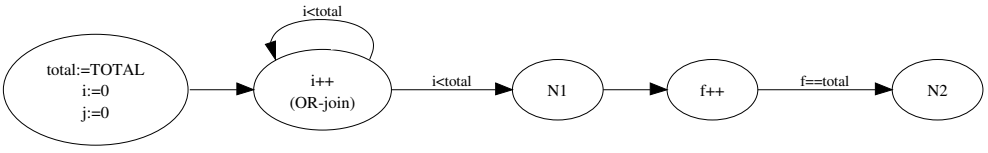

Figure 6.16: Multiple Instances With a Priori Runtime Knowledge (same token).

The first step in the graph from Figure 6.16, initialises the variables. The variable `total` repre-

sents the total number of activity instances to create. This value should depend on some domain data (`total:=TOTAL`, represents the computation of the initial value). The other two variables `i` and `f` represent the number of initiated and finished activity instances, respectively.

After initialisation, the workflow enters a cycle which spawns `N1` the preset number of times. After reaching the total number, there are no more available transitions from the second step. Whenever an instance of `N1` finishes, the finished activity instances counter is incremented. The condition in the final control flow to N2 guarantees that the execution of the workflow only proceeds after all instances of `N1` have finished.

All instances of `N1` are executed with the same token. As mentioned before, in another pattern (Section 6.12), it may be desirable to instantiate `N1` with different tokens. The solution shown in Figure 6.17 fulfils that requirement.

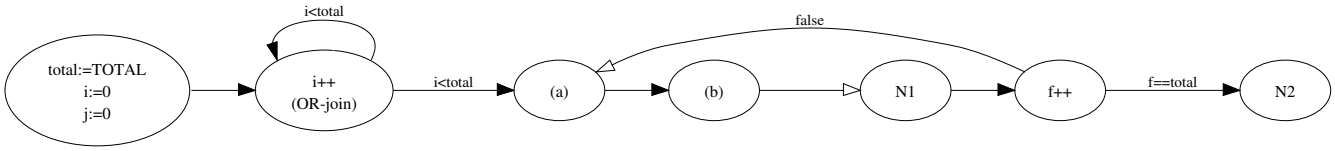

Figure 6.17: Multiple Instances With a Priori Runtime Knowledge (different token).

Step `(a)` is the target of a loop control flow. Its purpose is to store the loop exit token. Note that the flow leaves the loop only when `f == total`. However, the loop control flow pointing back to `(a)` is never executed, because its condition is `false`: It only serves to force storing the loop exit token at `(a)`. Step `(b)` is used to generate a new token. It is connected to `N1` by a loop control flow for that reason.

6.15 Multiple Instances Without a Priori Runtime Knowledge

Description An activity is enabled multiple times in the same workflow instance. The number of instances of the given activity is not known during design time, nor is it known at any stage during runtime before the instances of that activity have to be created. When all instances are completed some other activity needs to be started. While some of the instances are being executed or already completed, it should be possible to create new ones.

Solution At runtime, the engine needs to decide whether to create an instance of `N1`. The decision is modelled like in the Multiple Instances Without Synchronisation pattern (Section 6.12), using the activity `more?`, which computes some domain-specific decision. As before, variables `i` and `f` represent the number of initiated and finished activity instances, respectively. They are incremented immediately before and after the activity's execution.

It is necessary to consider the possibility of creating new instances of `N1`, both before and after each execution of `N1`. That is why there are two control flows going back to the `more?` step. With this construction it is possible to start another instance of `N1` even after all previously spawned instances have terminated, but still before running `N2`.

`N2` is only started when no more `N1` instances are desired and there are as many finished activities

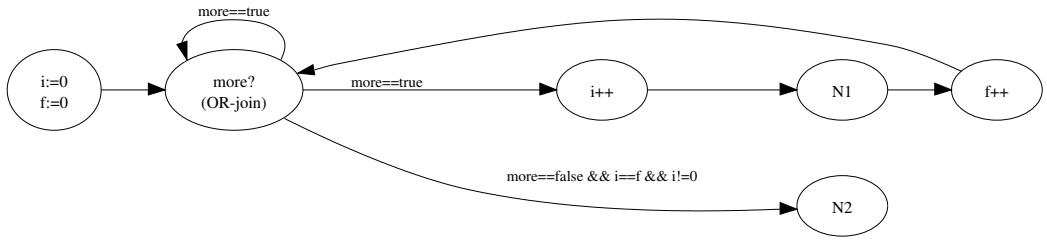

Figure 6.18: Multiple Instances Without a Priori Runtime Knowledge (same token).

as there are started. The condition $i\ !=\ 0$ implies that at least one instance of `N1` must have been created before `N2` can be executed. This restriction can be dropped, which enables the workflow instance to proceed to `N2`, with zero instantiations of `N1`.

Note that there is a race condition that might cause erroneous behaviour. Consider a situation where `more?` evaluated to `true` three times and then evaluates to `false`. If, due to concurrent scheduling, one instance of `i++`, `N1`, and `f++` have occurred, then `more==false && i==f && i!=0`, which means that `N2` will execute before all instances of `N1` have finished. The solution is to collapse the steps `more?` and `i++` together. Of course, the increment should only be executed when `more?` evaluates to `true`.

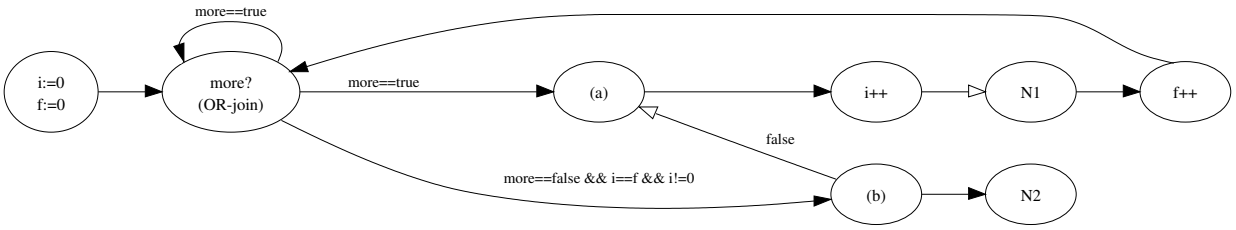

Figure 6.19: Multiple Instances Without a Priori Runtime Knowledge (different token).

Like in previous patterns involving multiple instances that are executed over the same structure of the model, we can provide a solution that uses different tokens for each instantiation of `N1`. This is shown in Figure 6.19. We have used the same strategy, as in the previous pattern. Step `(a)` stores the loop exit token, that is used when traversing from `(b)` to `N2`. We achieve this by making `(a)` the target of a loop control flow originating from `(b)`. Like before, this flow is never navigated, but causes step `(a)` to store the token on its first execution. Additionally, the flow that targets `N1` has been changed to a loop control flow to generate a new token.

6.16 Deferred Choice

Description A point in the workflow where one of several branches is chosen. The choice is not made explicitly like in the Exclusive Choice pattern (Section 6.4), but several alternatives are offered to the environment. However, in contrast to the Parallel Split pattern (Section 6.2), only one of the alternatives is executed. This means that once the environment activates one of the branches the

other alternative branches are withdrawn. The choice is delayed until the processing in one of the alternative branches is actually started, i.e., the moment of choice is as late as possible.

Solution The Deferred Choice pattern is not supported by the current back-end language. To make this pattern available, the meta-model needs, at least, to support event-based modelling, i.e., there must be a way to react to an external event.

Consider the following example: At a certain point in a workflow, a person needs to contact some company to order some goods. To do that she can either send an e-mail or a fax. However, she should not send the same request twice (i.e. use both communication channels). Moreover, the decision on which communication channel to use is not up to the workflow engine. The person should have both options available in her worklist and when she selects one of them to execute, the other should be removed. This is a typical case for applying the Deferred Choice pattern. In fact, we understand that until the person makes the choice of which activity to execute, neither has effectively begun, from the domain point of view. Thus, it should be possible to simply remove the other activity from the worklist, after the choice is made. But that cannot be done, because the engine cannot tell the difference between an activity placed on a worklist from an activity taken from it to be executed. In both cases, the activity has already started, from the engine's point of view.

On the one side, there is no cancellation mechanism in the model, to stop activities that are already executing and, on the other side, the engine has no knowledge of the step's internal state.

The first solution that we can think of is to change the process to adapt it to what the language supports. Generally, we could create an activity where the choice was made explicitly, and then, based on the choice the engine would go for the appropriate activity. In our example, the person would only have one work item, whose purpose was to decide which way to send the request. After this first work item was executed, only one of the two choices would appear in the worklist. This solution is the best that can be realised with the current back-end language and, still, it does not allow for the choice to be as late as possible. The choice must be implicitly made in the moment the person picks one of the activities to perform.

We would like to keep the current execution semantics, and still support this pattern (cancellation of one of the activities). A solution that maintains the engine's abstraction consists in allowing the activity to throw events. The engine doesn't need to understand the semantics of the events. It only needs to match the events thrown, with the events modelled in the back-end language. The workflow model would have the description of the event and a step to execute (which, in this case, would serve to cancel the other activity).

This mechanism can be generalised to support any kind of events. It would be simple to model a step that depended on an event generated by some external system; in turn, this step would propagate the event to the workflow model and some action could be taken.

This solution effectively supports the Deferred Choice pattern in a way that is compatible with the current execution semantics. In the case of cancelling an already started step, the step instance would simply disappear as if it had never been started. The cancelling step would, of course, have to understand the internal states of a step (see the Cancel Activity pattern, Section 6.19).

There is still a race problem if both activities are chosen concurrently. We have to take caution not to cause concurrent cancellation of all the activities, nor to allow continuation of more than one. This can easily be solved by supporting synchronous event handling. The step generating the event only proceeds after the event is successfully handled.

6.17 Interleaved Parallel Routing

Description A set of activities is executed in an arbitrary order: Each activity in the set is executed, the order is decided at runtime, and no two activities are executed at the same moment (i.e. no two activities of the set are active for the same workflow instance at the same time).

Solution There are basically two choices to implement this pattern. One takes more space to store and the other consumes more time to execute. The former consists in generating all the sequential combinations and enclosing them between the Exclusive Choice (Section 6.4) and the Simple Merge (Section 6.5) patterns. The `randomise` activity outputs a variable (`choice`) with a random value identifying the path to take. For each workflow instance a random path is chosen and no two activities are executed in parallel. Figure 6.20 shows this approach for interleaved Parallel Routing of three activities.

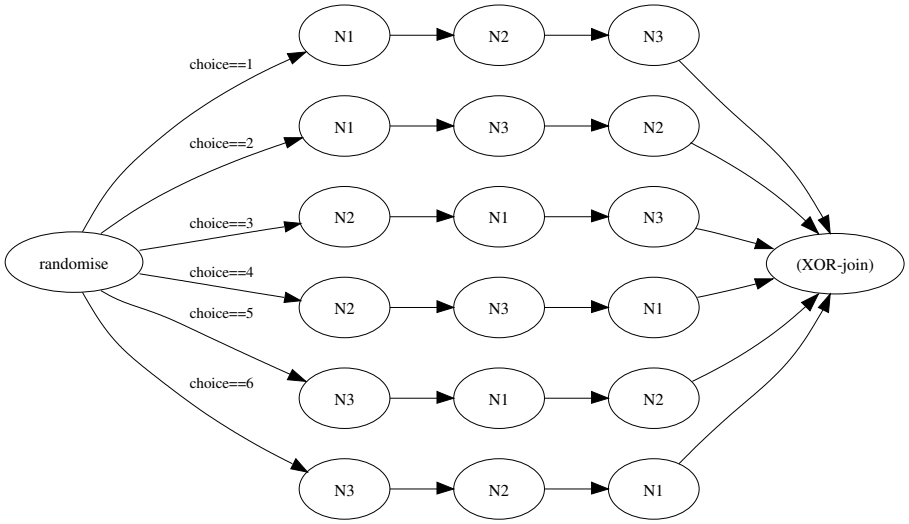

Figure 6.20: Interleaved Parallel Routing (enumerate all sequences).

The alternative consumes more time to execute, but it takes considerably less space to model and it doesn't repeat the declaration of the activities. It uses a loop (see Figure 6.21) and in each iteration it randomly chooses a path to execute. It must guarantee that each path is executed exactly once. So, the `randomise` step stores an array, which keeps track of the random choices made. The output of that step is the `choice` variable with a unique control flow identifier, e.g. a number, for the chosen control flow. The control flow conditions are exclusive and cover all possible values for the `choice`. We add another distinct value, e.g. zero (0), to represent that all activities have been executed and workflow execution should continue forward. The loop control flow from step `(a)` to the XOR-join step, serves to cause the propagation of the loop exit token stored at the `randomise` step when flow

leaves this structure.

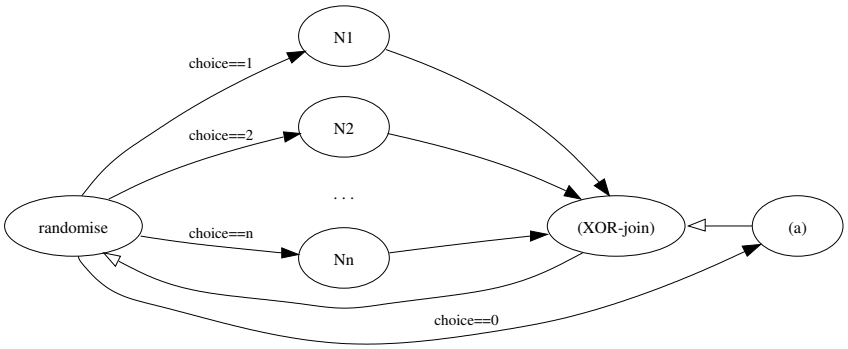

Figure 6.21: Interleaved Parallel Routing (iterative solution).

6.18 Milestone

Description The enabling of an activity depends on the workflow instance being in a specific state, i.e., the activity is only available if a certain milestone has been reached and it did not expire yet. Consider three steps named `N1`, `N2`, and `N3`. `N2` is only available after `N1` has been executed and before `N3` begins executing, i.e., `N2` is not available before the execution of `N1` nor after the execution of `N3`. The milestone corresponds to the time between ending `N1` and starting `N3`, where `N2` is available.

Solution Figure 6.22 shows a possible implementation for this pattern. There are two paths taken upon the termination of activity `N1`: The path to the optional execution of `N2` and the path to `N3`. The step `doN2?` represents a domain-specific choice whose output is whether `N2` should be executed. `N3` has to wait for this decision even if `N2` is not executed.

If the back-end language already supported the Deferred Choice pattern (Section 6.16), then it would be possible for the execution of `N3` to cancel the execution of `doN2?` and/or `N2`. This way, the execution of `N3` would not need to be kept "on hold" until the decision whether to execute `N2` was taken.

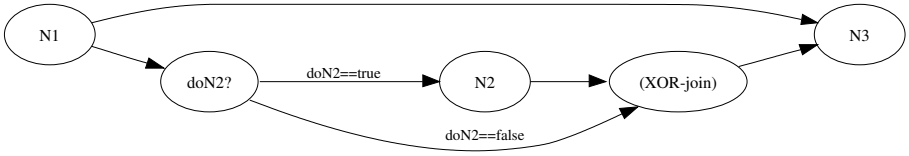

Figure 6.22: Milestone with a single execution.

From the pattern's description it is not clear whether activity `N2` can be executed multiple times while the workflow execution is between `N1` and `N3`. We can support this with a small change in the model. In Figure 6.23 we basically changed `N2`'s outgoing control flow to point back to `doN2?`.

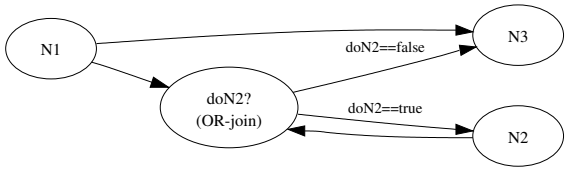

Figure 6.23: Milestone with multiple executions.

6.19 Cancel Activity

Description An enabled activity is disabled, i.e. a thread waiting for the execution of an activity is removed.

Solution There are two options to implement this pattern. The activity can be cancelled in an asynchronous or in a synchronous fashion. In the first case, we need the functionality described in the Deferred Choice pattern (Section 6.16), which requires support for event-based modelling. In the second case, the compilation process simply generates a step in the back-end that when executed removes another instance from execution. In both cases, the cancelling activity uses the engine's API to remove another activity instance from execution. The cancelling activity must be aware of the step's internal states (at least differentiate between `ACTIVATED` and `STARTED`[7]).

The first case implies support that is currently inexistent in the back-end language. The advantage is that it would become possible to change the workflow behaviour depending on external events.

Either solution becomes more complex if we consider that in the same workflow instance there might be more than one instance of the same activity. This means that either all instances are cancelled or there must be a way, in the model, to identify the specific instance to cancel.

6.20 Cancel Case

Description A case, i.e., a workflow instance is removed completely. Even if parts of the workflow are instantiated multiple times, they are all removed.

Solution The behaviour described in this pattern is exactly what happens when the workflow execution reaches its final step. The final step, which is identified as such in the workflow model, represents an explicit termination of the workflow execution. In the most common circumstances, when this last step is reached, there should be no more activity instances. However, if there are any, they will be terminated. If one wants to model the termination of the workflow instance at any point, it suffices to make a control flow to the last step, which should have an adequate synchronisation rule.[8]

[7] An ACTIVATED step hasn't yet performed any domain-specific work, although it has already begun, from the engine's point of view. A STARTED step, on the other hand, is already performing domain-specific work. As as example consider a work item: As soon as it is placed on a worklist it is already ACTIVATED, but not yet STARTED.

[8] Note that the default AND-join might block the end of the workflow if there are many control flows targeting the end step.

6.21 Comments

In this chapter we analysed the expressiveness of the back-end language using the workflow patterns research as the guideline. These patterns cover far more than the common patterns supported by most workflow languages. One important fact should be noted: The authors of the study on the workflow patterns evaluated many WfMSs and they considered that these systems didn't support many of the patterns described. This is because they focused on whether there was "native" support in the model for representing such patterns. Consider for example the Interleaved Parallel Routing pattern (Section 6.17). In our back-end language this pattern is implemented by construction, i.e., there is no specific construct to model the pattern (it needs to be "programmed" using auxiliary steps). According to the author's criteria, this pattern is not supported by our language, even though it can be modelled. Off course, this is true, but our point is that in their criteria the authors assume that these languages are to be directly manipulated by the workflow modellers, and that they should not have to model complex constructions to represent their ideas. This assumption does not hold for the back-end language, which is not supposed to be used directly by any user. Expressiveness of the front-end languages is irrelevant in the context of our evaluation. Our goal with this evaluation is to demonstrate that our back-end language is suitable for modelling workflow patterns, and/or to identify possible lacks of expressiveness. We are satisfied as long as the back-end language can realise a pattern, be it by construction or natively. Ideally, we want to have a language with a set of constructs as minimal as possible. In our view this facilitates the development of new functionalities in the WfMS, because the simpler the semantics of the back-end language, the less error-prone it will be to extend the WfMS. Still, we do not wish to complicate the compilation of a front-end language. By providing a description of the patterns' possible implementations we are also guiding the developers of front-end languages in the process of compiling them to the back-end.

From the evaluation presented in this chapter we understand that the back-end language already provides a good coverage for the workflow patterns. Perhaps the most noted lack of support was in the Deferred Choice pattern (Section 6.16). We have proposed a possible extension that would provide the missing support: To provide event-based modelling (synchronous and asynchronous) in the back-end language. This includes the possibility of throwing events and the possibility of modelling behaviour in response to them. In the patterns at study, the events could be used to provide cancellation mechanisms useful to the implementation of the following patterns: Deferred Choice (Section 6.16), Milestone (Section 6.18), and Cancel Activity (Section 6.19).

7 Concluding Remarks

This chapter summarises the main results achieved and presents possible directions for future work.

7.1 Results

Model for a *Workflow Virtual Machine*. First and foremost, this thesis is based on the identification of a problem that embraces WfMSs: The survival of the system throughout many changes to its workflow definition language. We faced this problem ourselves when developing the system that initially preceded WorkSCo. The definition of the WfVM has provided us with:

- The ability to change/extend the front-end language without affecting the WfVM. We have successfully added new features to WorkSCo's front-end language without having to change the underlying WfVM.
- The ability to enhance the system's functionalities in a (front-end) language-independent fashion.

Moreover, the WfVM solves the conflicting forces between workflow modellers and system developers.

Different domains have different needs. The study of other systems and languages has taken us to believe that one of the main reasons for such a high number of workflow languages, besides industry competition over standards, is the fact that workflow languages tend to be adapted to suite specific needs, i.e., it seems to be important to provide domain-specific modelling concepts to the workflow modeller. Therefore, the search for a universal modelling language appears to be a conflicting thought. We have addressed this problem the other way around: We accept the existence of many different languages and try to minimise their impact on the system. A system that can support many languages with little effort, has a stronger chance of surviving.

Back-end language. We have proposed a concrete back-end language to implement the WfVM. This language needs to be expressive enough so that it can model the workflow concepts from several front-end languages. We have investigated this issue by doing a patterns-based analysis of the language. We have shown that the majority of the patterns are supported either natively or by construction. A few patterns require adding code to the steps, such as the Deferred Choice pattern. This situation would pose a modelling problem in other WfMSs, where the model that is designed is the same model that is instantiated and executed by the engine. In our case, the compilation process has the freedom to add steps to the back-end definition, thus supporting any missing features.

Front-end language. The front-end language we implemented is just one of many possible languages. What is relevant here is that, after having a WfVM in place, the construction of the front-end language was straightforward. The decision of which concepts to support was driven solely by the domain modelling requirements. The process of supporting the language in the WfVM was a matter of defining the compilation rules and creating the required implementations for the step's activities.

WfMS. As a final result we have put together the core functionality of a usable open source WfMS, based on the concept of the virtual machine and the front-end language that we developed.

7.2 Future Work

Implement support for more front-end languages. The concrete back-end language that we presented is sustained both by the support it provides to our front-end language and its capacity to support almost all of the workflow patterns we tested. Still, we consider that it would be interesting to experiment compiling more front-end languages to this back-end. We consider of special interest to support some of the most well-known standards proposals, such as BPEL and XPDL. More than prove our point, it could increase the interest and usefulness of WorkSCo.

Create workflows composed from different languages. Although this thesis did not address this problem, it opened the way for a WfMS that can execute workflows created by composing other workflows that are modelled in different languages. We already have an approximation to this concept. Our front-end language has a `SubWorkflowProcedure`, which runs a workflow given its unique identifier. It does not know which front-end language produced it.

Support the remaining patterns. The study conducted on the workflow patterns applied to the back-end language has suggested that the next most useful feature would be to include support for event-based modelling and event-based execution in the WfVM. This would facilitate the support to patterns such as the Deferred Choice as it was mentioned in Section 6.16.

Implement API for generating back-end patterns. Currently the process of compiling any front-end language is responsible for generating every aspect of the back-end language. This could be simplified. Taking the workflow patterns, we could implement a layer that provides an API for constructing predefined workflow patterns. This layer would then be used by the compilation process of any front-end language, whenever it was necessary to a generate any of the supported patterns. This concept can be extended to the informational perspective, as well.

Acronyms

BPEL *Business Process Execution Language*.

FOP *Feature Oriented Programming*.

OIS *Office Information System*.

WfDL *Workflow Definition Language*.

WfES *Workflow Enactment Service* (the core component in the WfRM).

WfMC *Workflow Management Coalition*.

WfMS *Workflow Management System*.

WfRM *Workflow Reference Model* (from the WfMC).

WfVM *Workflow Virtual Machine*.

WorkSCo *Workflow with Separation of Concerns*.

WSFL *Web Services Flow Language*.

B Bibliography

[AADH04] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and implementation of the YAWL system. In A. Persson and J. Stirna, editors, *Advanced Information Systems Engineering, Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04)*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159. Springer-Verlag, Berlin, 2004. `http://is.tm.tue.nl/research/patterns/download/caise_yawl_2004.pdf`.

[AAF+02] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, Ivana Trickovic, and Sinisa Zimek. *Web Service Choreography Interface (WSCI) 1.0*. World Wide Web Consortium, 2002.

[Aal03] W.M.P. van der Aalst. Don't go with the flow: Web services composition standards exposed, 2003.

[ACD+03] 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*. BEA, IBM, Microsoft Corporation, SAP AG, Siebel Systems, May 2003.

[ACE] Adaptable and Composable E-commerce and Geographic Information Services (ACE-GIS). `http://www.acegis.net`.

[AH05] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005. `http://is.tm.tue.nl/research/patterns/download/yawl_qut_report_FIT-TR-2002-06.pdf`.

[AHKB00] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer-Verlag, Berlin, 2000. `http://is.tm.tue.nl/research/patterns/download/coopis.pdf`.

[AHKB03] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003. `http://is.tm.tue.nl/research/patterns/download/wfs-pat-2002.pdf`.

[Ark02] Assaf Arkin. *Business Process Modelling Language*, November 2002. `http://www.bpmi.org/BPML.htm`.

[AS94] Kenneth R. Abbott and Sunil K. Sarin. Experiences with workflow management: issues for the next generation. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 113–120, New York, NY, USA, 1994. ACM Press.

[Bae04] Tom Baeyens. The State of Workflow. `http://jbpm.org/state.of.workflow.html`, May 2004.

[Bat] Don Batory. The ahead tool suite. `http://www.cs.utexas.edu/users/schwartz/ATS.html`.

[Bat04] Don Batory. Feature-oriented programming and the ahead tool suite. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 702–703, Washington, DC, USA, 2004. IEEE Computer Society.

[BKKR03] Martin Bernauer, Gerti Kappel, Gerhard Kramler, and Werner Retschitzegger. Specification of interorganizational workflows — a comparison of approaches. In *7th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2003)*, pages 30–36, Orlando, USA, July 2003.

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium, March 2001.

[CKO92] Bill Curtis, Marc I. Kellner, and Jim Over. Process modeling. In *Communications of the ACM*, volume 35. Association for Computing Machinery, September 1992.

[COM] COMponent-Based INteroperable Enterprise system development (COMBINE). `http://www.opengroup.org/combine/overview.htm`.

[DLTW97] Ann DiCaterino, Kai Larsen, Mei-Huei Tang, and Wen-Li Wang. *An Introduction to Workflow Management Systems*. Center for Technology in Government, University at Albany, November 1997.

[Dom05] Dulce Domingos. *Controlo de Acesso em Fluxos de Trabalho Adaptáveis*. PhD thesis, University of Lisbon, Lisbon, Portugal, 2005. Written in Portuguese.

[DS99] Weimin Du and Ming-Chien Shan. Enterprise workflow resource management. In *RIDE '99: Proceedings of the Ninth International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises*, page 108, Washington, DC, USA, 1999. IEEE Computer Society.

[DWF] DWFMS. `http://co-operate.inescporto.pt:8081/ours2`.

[Ell79] C.A. Ellis. Information control nets: a mathematical model of office information flow. In *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems*, pages 225–240, Boulder, CO, USA, 1979. ACM Press.

[EN80] Clarence A. Ellis and Gary J. Nutt. Office information systems and computer science. *ACM Comput. Surv.*, 12(1):27–60, 1980.

[FCS04] Sérgio Fernandes, Joäo Cachopo, and António Rito Silva. Supporting evolution in workflow definition languages. In Peter Van Emde Boas, Jaroslav Pokorný, Mária Bieliková, and Július Štuller, editors, *Proceedings of the Conference on Current Trends in Theory and Practice of Computer Science*, volume 2932 of *Lecture Notes in Computer Science*, pages 208–217, Měřín, Czech Republic, 2004. Springer-Verlag.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GHS95] Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases*, 3(2):119–153, 1995.

[Hol95] David Hollingsworth. *The Workflow Reference Model*. Workflow Management Coalition, Jan 1995. Document Number TC-00-1003.

[jBP] jBPM. `http://www.jbpm.org/3/`.

[Kie02] Bartosz Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2002.

[Lea99] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, second edition, 1999.

[Ley01] Frank Leymann. *Web Services Flow Language (WSFL 1.0)*. IBM Corporation, May 2001. `http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.

[LS97] Yu Lei and Munindar P. Singh. A comparison of workflow metamodels. In *ER-97 Workshop on Behavioral Modeling and Design Transformations: Issues and Opportunities in Conceptual Modeling*, Los Angeles (CA), 1997.

[MÖ99] Michael zur Mühlen. Evaluation of workflow management systems using meta models. In *HICSS '99: Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences-Volume 5*, page 5060, Washington, DC, USA, 1999. IEEE Computer Society.

[MÖ04] Michael zur Mühlen. *Workflow-based Process Controlling. Foundation, Design, and Implementation of Workflow-driven Process Information Systems.*, volume 6 of *Advances in Information Systems and Management Science*. Logos, Berlin, 2004.

[Man01] Dragos-Anton Manolescu. *Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[McC92] S. McCready. There is more than one kind of workflow software. *Computer World*, 1992.

[Moh96] C. Mohan. Tutorial: State of the art in workflow management system research and products. In *Tutorials of ACM SIGMOD International Conference on Management of Data(Booktitle)*, Tutorials of ACM SIGMOD International Conference on Management of Data(Series), 1996.

[Nut96] Garry Nutt. The evolution toward flexible workflow systems. *Distributed Systems Engineering*, 3(4):276–294, 1996.

[OMG00] Object Management Group. *Workflow Management Facility Specification, Version 1.2*. OMG, 2000.

[ope] Open Source Workflow Engines in Java. `http://www.java-source.net/open-source/workflow-engines`.

[OSW] OSWorkflow. `http://www.opensymphony.com/osworkflow/`.

[SZ01] Edward A. Stohr and J. Leon Zhao. Workflow automation: Overview and research issues. *Information Systems Frontiers*, 3(3):281–296, 2001.

[Tha01] Satish Thatte. *XLANG — Web Services for Business Process Design*. Microsoft Corporation, 2001. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`.

[Top] Topicus BV. The open source workflow initiatives. `http://www.gripopprocessen.nl/index.php?id=35`.

[Twi] Twister. `http://www.smartcomps.org/twister/`.

[wfm] Workflow Management Coalition. `http://www.wfmc.org`.

[WMC98a] Workflow Management Coalition. *Audit Data Specification*, 1998. `http://www.wfmc.org/standards/docs/TC-1015_v10_beta.pdf`.

[WMC98b] Workflow Management Coalition. *Workflow Client Application Application Programming Interface (Interface 2 & 3) Specification*, 1998. `http://www.wfmc.org/standards/docs/if2v20.pdf`.

[WMC99a] Workflow Management Coalition. *Workflow Management Coalition Terminology & Glossary*, February 1999. `http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf`.

[WMC99b] Workflow Management Coalition. *Workflow Standard—Interoperability Abstract Specification*, 1999. `http://www.wfmc.org/standards/docs/TC-1012_Nov_99.pdf`.

[WMC02] Workflow Management Coalition. *Workflow Process Definition Language—XML Process Definition Language*, 2002. `http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf`.

[Wor] Workflow with separation of concerns. `http://worksco.sourceforge.net`.

[YAW] YAWL. `http://www.yawl.fit.qut.edu.au/`.

[Zis77] Michael Zisman. *Representation, Specification and Automation of Office Procedures*. PhD thesis, University of Pennsylvania, 1977.