WebWorkFlow
a Workflow Modeling Language for Web Applications

Version of March 20, 2009

Ruben Verhaaf

WebWorkFlow
a Workflow Modeling Language for Web Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ruben Verhaaf
born in Ede, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

WebWorkFlow
a Workflow Modeling Language for Web Applications

Author:	Ruben Verhaaf
Student id:	1153749
Email:	`R.Verhaaf@gmail.com`

Abstract

Workflow is relevant for any business that wants to have IT support for their recurrent business processes. The World Wide Web is increasingly becoming a platform for which full-scale business applications are developed. However, the two paradigms are altogether different. To provide a better way of developing workflow-enabled web applications, we design WebWorkFlow, a workflow modeling language for web applications. It features complete generation of workflow-enabled web applications. WebWorkFlow is an extension of WebDSL, an existing domain-specific language for the web. WebWorkFlow provides *procedures* for specifying tasks with an optional *user interface*, to be performed by *actors* under certain *conditions*. It provides a *process language* for specifying the control-flow of workflow processes in which procedures are used. The separation of procedures and processes leads to high *flexibility*, while the relatively simple process language is *expressive*. We evaluate WebWorkFlow's coverage by implementing workflow patterns. We perform a case study by implementing the LIXI Property Valuation workflow at NICTA, Australia.

Thesis Committee:

Chair:	Prof.dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor:	Drs. Z. Hemel, Faculty EEMCS, TU Delft
Company supervisor:	Dr. L. Zhu, NICTA Sydney
Committee member:	Prof.dr.ir. G.J.P.M. Houben, Faculty EEMCS, TU Delft
Committee member:	Dr.ir. R. van Solingen, Faculty EEMCS, TU Delft

Preface

I am extremely thankful to the following people. Eelco Visser, my supervisor at DUT, for providing me with a great research project and guiding me through the process of finding my way in it. Liming Zhu, my supervisor at NICTA, for his guidance and comments on my thesis work. Zef Hemel, for frequent guidance and invaluable help during the implementation of WebWorkFlow. Shirley Xu, for her help in demonstrating and explaining the LIXI valuation system. To the other WebDSL developers Danny Groenewegen, Sander Vermolen, and Lennart Kats for giving me a nice place to work. To the researchers at NICTA for giving me a warm welcome and being great hosts: Mark Staples, Paul Bannerman, Paul Mackie, Ross Jeffery, Udo Kannengiesser, and Kevin Lee. To Prof.dr. Arie van Deursen and Prof.dr.ir. Geert-Jan Houben, for taking part in my graduation committee. To Dr.ir. Rini van Solingen for taking part in my graduation committee and for introducing me at NICTA.

Warm thanks also to my parents, brother, sister, niece, and many great friends for supporting me from home while I was in Sydney.

Ruben Verhaaf
Delft, the Netherlands
March 20, 2009

Contents

List of Figures

Chapter 1

Introduction

All businesses involve *business processes*, that are often supported by IT-systems running *workflows*. A *business process* can be thought of as any activity in the context of a company, ranging from a simple procedure like locking the door after closing a shop and leaving the key in the mailbox, to a complicated process like organizing an academic conference. *Workflow* is concerned with the modeling of business processes and thereby improving efficiency, reliability and transparency of these processes. Workflow involves *tasks*, being units of work, performed by workflow *participants*. These tasks are scheduled and performed according to a *process description*, involving sequencing and parallel execution of tasks.

Since the advent of the *World Wide Web* [3], its popularity has increased to the point where many businesses can barely function without internet connectivity. After business application models have shifted from terminal-operated applications running on a mainframe computer to applications running on a *personal computer* in the 1980s, the central paradigm for business applications is again shifting, to a model in which applications run on a server connected to the internet, and are operated using a web interface. Some characteristics of the World Wide Web (WWW) are noteworthy. The central notion in the internet is that of a *resource* [19], which can be a document or page, or an image. Using *hyperlinks*, it is possible to connect resources to form a web of resources. For the internet, the *HyperText Transfer Protocol* (HTTP) is used to manage requests for a resource. Using HTTP, a request for a resource is made by a client, after which the server responds by sending the intended resource. HTTP is a *stateless* protocol, in which each request is treated as an independent transaction.

Following the trend of web-based business applications, we want to look at the development of a web-based workflow system. Developing a process-oriented application on the internet, however, is non-trivial. Although hyperlinks can be used to provide some sort of sequencing for pages, a workflow application demands more control over the scheduling and execution of tasks. Therefore, it is necessary to specifically design a system to handle the performing of tasks and the scheduling of tasks according to process descriptions on the internet. The most widely used workflow solution for the web currently is BPEL4WS, the *Business Process Execution Language for Web Services*. It focuses on the process perspective and on the integration of heterogeneous systems, using web services as an interface between different systems. With this solution, it is tedious to implement a fully working workflow system for web-based use, as every task has to be separately defined and made available using a web service interface. We want to develop a solution that enables complete generation of applications, in which both the process description and the implementation of individual tasks can be specified using a single language. Also, user interfaces for task execution must be fully customizable.

Many workflow languages provide a large amount of constructs for the process perspective. However, a developer may want to model a pattern that is not directly supported using the abstractions provided. In this case, a problem arises, as it is not normally possible to devise custom constructs. Moreover, the workflow description language does not provide any possibilities to use a lower level of abstraction. As a consequence, the developer will need to use the available abstractions to model the intended pattern, which may cost a considerable amount of effort and introduce additional complexity into the process model. Additional flexibility is desirable.

Seeking to improve on the aspects of workflow systems mentioned above, we endeavour to create a language that provides the following:

- Process-oriented abstractions for web applications, involving the notions of *tasks*, *participants* and a *process description*. The majority of regularly used control-flow patterns must be covered in the process description.
- Complete generation of applications, in which both the process description, the implementation of individual tasks and a custom user interface can be specified using a single language.
- Flexibility in respect to the scheduling of workflow tasks. This involves having multiple abstraction layers with the process specification at the highest level, while making available the constructs of lower abstraction layers within higher layers.

Focusing on these goals, we do not consider the integration of external heterogeneous systems and leave this issue for future work.

In the remainder of this chapter, we will define research questions and an approach to formulate an anwer to these questions.

1.1 Research Questions

The research questions investigated in this thesis are as follows:

1. What is a good design for a language that fulfills the requirements stated above?
2. How to implement such a language?
3. How does this language work in practice?

1.2 Approach

We will address these research questions in the rest of this thesis. In this section, we will concisely sketch the solutions we have arrived at. We will do so by first introducing the language design of

WebWorkFlow, a workflow modeling language for web applications. Next, we will describe the method used to implement this language. Then we will describe our approach to evaluating the usefulness of this language.

1.2.1 Language Design

We have developed *WebWorkFlow*, a language containing abstractions for process-oriented websites. To maximize flexibility in respect to the process perspective of workflows, we split the specification of workflows into two different layers of abstraction: the *procedure layer* and the *process layer*.

The *procedure layer* is concerned with defining atomic tasks on *objects*. A procedure consists of a specification of the *actor* that can perform the procedure, the *conditions* under which the procedure can be performed, an optional *custom user interface* for the procedure and an *action* that specifies what is actually done when the procedure is performed. It is extended with an *event model* to insert code to perform when a procedure is *enabled*, *done*, or *disabled*.

The *process layer* is positioned on top of the procedure layer and introduces the *composition* of procedures using a *process description*. An additional event handler is introduced for use after a composite procedure has been *processed*.

This organization introduces flexibility by enabling the developer to express the most commonly used patterns using a *process description*, while keeping the option open to define *procedure-level* amendments to the procedure-level code generated from the process descriptions.

1.2.2 Method

We have implemented WebWorkFlow as several layers of abstraction on top of an existing language for web applications called *WebDSL* [70].

WebDSL features the definition of *pages* and *entities*, extended with *templates* and a sub language for *action control*. Using a partial compilation of WebWorkFlow code to WebDSL code, we effectively implement a web-based application from a workflow description.

As WebDSL page definitions support defining a completely *custom user interface*, we can use WebDSL's constructs to incorporate the possibility for defining custom user interfaces in WebWorkFlow by allowing the developer to use basic WebDSL code in the definition of procedures, in the same way procedure-level constructs can be used in the process level.

For the implementation of WebWorkFlow, we extend the implementation of WebDSL, which is implemented using Stratego/XT [8], a toolset for model-driven development using *rewrite rules* combined with programmable *rewriting strategies*. As Stratego exhibits both *vertical* and *horizontal modularity*, it allows for implementing the additional layers of abstraction needed for WebWorkFlow using a separate plugin.

1.2.3 Evaluation

To evaluate WebWorkFlow's coverage, we will look at 40 different patterns given by Van der Aalst et al. in [61, 55, 52, 53] and implement these patterns for WebWorkFlow if possible.

We will also evaluate the usefulness of WebWorkFlow by performing a case study performed at NICTA (National ICT Australia), located in Sydney, Australia. In this case study, a workflow for valuation of real-estate property used in the context of the LIXI project (Lending Industry's XML Initiative), is modeled using WebWorkFlow. We will use the experience gained to evaluate WebWorkFlow. Also, we will look at the size of the workflow definition code used in the Valuation Workflow and compare it with the size of the rest of the code.

To evaluate WebWorkFlow as a DSL, we will assess its merit in the face of the following evaluation criteria: *expressivity*, *coverage*, *completeness*, *portability*, *code quality*, and *maintainability* [69].

1.3 Outline

In this thesis, we will first explore the theoretical background of the domains of workflows and domain-specific languages in chapter 2. In chapter 3, we will explain the different elements of WebDSL, on which we base our workflow solution. We will present WebWorkFlow in chapter 4, giving two examples of workflow implementations. In the two following chapters, we will first explain the syntax and inner workings of procedures and their transformation to WebDSL code, and subsequently the transformation of process descriptions to procedure-level code. In chapter 7, we perform our case study with the LIXI valuation workflow, a workflow that is actually used in the industry. Next, we will explore coverage of our language by implementing the workflow patterns provided by Van der Aalst et al. in [61, 52, 53]. We will then look at related work and compare this with WebWorkFlow in chapter 9. In chapter 10, we will evaluate the results of this research project. In the final chapter, we will present our contributions, draw conclusions from our work and discuss several ideas for future work.

Chapter 2

Background

Workflow is concerned with the modeling of *business processes* and thereby improving efficiency, reliability and transparency of these processes.

A *business process* can be thought of as any activity in the context of a company, ranging from a simple procedure like locking the door after closing a shop and leaving the key in the mailbox, to a complicated process like organizing an academic conference. As a rule, workflows are designed for repetitively performed activities, especially in the field of everyday administration. Often, a workflow revolves around a document or file, that is created or altered in the process of performing the workflow.

Domain-specific languages (DSLs) are languages that are developed especially for use in a certain domain. This is different from so-called *general-purpose languages*.

The field of domain-specific languages is related to the field of *model-driven architecture*, in which the focus of work from programming is moved to solution modeling, in order to "increase development productivity and quality by describing important aspects of a solution with more human-friendly abstractions and by generating common application fragments with templates." [57]

In the remainder of this chapter, we will first define workflows in more detail in section 2.1. After that, we will walk through a short history of the development of workflow systems in section 2.2. In section 2.3, we will address the characterization of workflows. In sections 2.4 and 2.5 we will list some of the challenges involved in developing a workflow management system and give some examples of workflows. After that, we will describe different types of DSLs in 2.6 and in 2.7, we will address several methodologies that can be used when designing DSLs.

2.1 Workflow: Definition and Terminology

Many different definitions of workflow have appeared in literature. WfMC, the Workflow Management Coalition, handles the following definition of workflow with an accompanying definition of a Workflow Management System (WFMS), that can execute workflows:

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

Workflow Management System: A system that 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.

This definition for workflow is very general. It is also tightly connected with the WFMS paradigm, in which a separate piece of software is used to execute workflows.

Manolescu [36] uses the following definition:

Workflow coordinates *activities* performed by various *participants* towards a business goal. [...] Coordination involves activity ordering and the interdependencies between them, synchronization with external events, and delegation to participants.

This definition is much more suited for our ends, as it does not imply a separate WFMS. In this definition, the notion of *participants*, performing the activities stands out clearly. *Activities* are equivalent to *tasks* in other definitions.

In [22], workflow is defined as follows:

Workflow: We define a workflow as a collection of tasks organized to accomplish some business process. [...] In addition to a collection of tasks, a workflow defines the order of task invocation or condition(s) under which tasks must be invoked, task synchronization, and information flow (dataflow).

This definition states more clearly what kind of interdependencies between tasks can be present. It does not mention the participants of a workflow, though.

We will use the following definition of workflow, derived from the definitions mentioned above.

Definition A *workflow* consists of *tasks*, performed by *participants* to accomplish some business goal. In addition to this, a workflow contains a *process description*, defining the order of task invocation, condition(s) under which tasks must be invoked and to what participants the tasks must be delegated.

Tasks can be atomic activities such as filling in a form, sending an email, or handing over a key. Tasks can also consist of groups of tasks, following a process description: a *subworkflow*.

The *participants*, or *actors*, that perform the tasks can be either specific persons, or persons that have a specific *role*, like administrator, valuer or manager. If the workflow is powered by an IT System, these human tasks can either include interaction with computers closely - by providing input commands - or loosely - by merely indicating the progress of tasks [22]. Another possibility is for the participant to be a software system. In this case, we speak of an *automated task*.

The *process description* forms the heart of the workflow. The actual ordering of events is directed by the description of the process. It also specifies conditions under which the tasks are to be performed.

2.2 History of Workflow Systems

Workflow systems have been developed since the 1970's. The development of workflow systems has had its ups and downs during this time. In this section, we will shortly describe the history of workflow system development, building on [16, 17].

According to [16], when development of workflow systems started in the 1970s, there was much optimism about the early workflow-related systems, then called *Office Information Systems (OISs)*. In the 1980s, the enthusiasm dropped and office automation was criticised by many, mainly because of the many failures of OISs during the 1970s. Much of the emphasis in the late 1980s was placed upon Computer-Supported Cooperative Work (CSPW) systems. In 1990, there was an increase in interest in workflow systems. Many companies developed their own proprietary workflow specification languages. The focus moved from workflow information systems to the idea of Business Process Reengineering, in which not only a company's information system, but also their internal business processes were looked at. This is also the period when attempts to standardize workflow solutions and languages were made by the Workflow Management Coalition (WfMC). Since 2000, on the level of executable workflows, the efforts focused on integrating different systems using a serviced architecture. Several combined efforts of companies resulted in BPMN and BPEL, which became de facto standards in respectively Business Process Modeling and Business Process Execution.

In the remainder of this section, we will discuss the different eras in workflow system development: *Office Automation*, *Computer-Supported Cooperative Work (CSCW)*, *Business Process Reengineering* and *recent developments*.

2.2.1 Office Automation

The development of workflow systems started as *Office Automation* with so-called *Office Information Systems*. In the early days of office automation, people were very optimistic about the expected results of office information systems. According to Ellis, however, much of this optimism was unfounded: "It was observed that organizations succeed only if people creatively violate, augment, or circumvent the standard office procedures when appropriate. When these electronic coordinators were introduced into offices, people could no longer blatantly disobey the office procedures. In many cases, these systems led to ineffective organizations and technology rejection. Thus, the rigid systems of the 1970s tended to interfere with work routines rather than expedite them" [16]. Another cause of failure was that technology for workflow systems was not yet widely available: not everyone in an office had a personal computer, and networking was not yet commonly used.

Office automation is about creating systems to automate offices. According to Ellis et al., *office* can be defined as "that part of a business that handles the information dealing with operations such as accounting, payroll, and billing" [17]. Any activities are usually triggered by the arrival of a request for service, such as an order or a bill. Ellis et al. mention several models that can be used by computer scientists to describe office activity:

- activities, resulting from requests for service, each requiring a supporting file system
- people carrying out tasks, communicating with and referencing a supporting file system

- communication media like a telephone, a file or an electronic message system and the use of these media
- a large database with users accessing and manipulating data.

Hirschheim has a more concise definition of an office: “Offices can be thought of as centres of organisational information handling and processing” [26, 71]. Office automation is also defined by Olson and Lucas as “the use of integrated computer and communication systems to support administrative procedures in an office environment” [46, 71]. These definitions all allow for both document-driven and process-oriented systems to be called office information system. Some early OISs are *Officetalk* and *SCOOP*.

Designed by William Newman and Tim Mott, Officetalk-Zero, or *Officetalk* for short, was implemented during 1976 and 1977, in an environment of multiple ‘minicomputers’, connected with a communication network [17]. It involved one of the first graphical user interfaces. The Officetalk network includes a file server for storing files, and maintaining a database. The system was designed as a distributed system that stored as much of the user state as possible in the shared file server, and only little information on the local computers. Officetalk was aimed at supporting document management, preparation, and communication and emphasizes the user interface. To develop a particular application for Officetalk, a set of blank forms must be designed, using the provided forms editor. Subsequently, these forms can be used for filling in and storing them on the file server. The communication system in Officetalk resembles email.

SCOOP, developed by Michael Zisman in 1977, has a different approach. He uses Petri-nets to model office procedures, rather than focusing on user interface and automation of office devices. In his modeling technique, a separate Petri-net is constructed for each *agent* in the procedure. These Petri-nets have the possibility to initiate other Petri-nets. It is also possible to use a high-level nonprocedural specifications language for defining documents, and activity details.

While Officetalk is mainly concerned with tools for office-related activities, SCOOP has a view on processes, instead of a simple collection of isolated tasks and pieces of equipment [17]. Officetalk can be seen as a document-driven system, whereas SCOOP is one of the first process-oriented systems.

2.2.2 Computer-Supported Cooperative Work (CSCW)

The field of Computer-Supported Cooperative Work, coming up in the 1980s, is broader than just workflow. We will shortly expound its characteristics.

Workflow systems, or process-oriented systems, are often focused on supporting the execution of a relatively rigidly defined process. However, not all processes can be defined rigidly. For instance, the process of designing a software system can hardly be poured into a well-defined workflow, because it is much less predictable than for instance the approval of a document. At best, a high-level overview of the process can be given, and software can be created to support for instance the filing of documents according to this high-level process description. While these systems are not exactly workflow systems according to our definition, they are closely related. Systems to aid developers or other people in an organization in a process in which collaboration is exploited to reach a common goal, is called *Computer-Supported Cooperative Work (CSCW)*. Sometimes, the term *groupware* is also used to describe this kind of systems.

Non-process oriented CSCW systems are called *project-oriented systems* by Mahling et al. [35]. They mention the trade-off that is often involved in choosing between process-oriented and project-oriented systems: control versus flexibility. A process-oriented system will maximize the control, while a project-oriented system will provide the users with more flexibility.

There are several classes of CSCW systems. Rodden [51] identifies four different classes: message systems, conferencing systems, meeting rooms and co-authoring and argumentation systems. A detailed description of these different classes is beyond the scope of this work.

Two examples of CSCW systems that were developed during the 1980s are *Poise* and *Polymer / D-Polymer*. In the early 1980s, the Poise (Procedure Oriented Interface for Supportive Environments) system was developed by the Collaborative Systems Laboratory [35]. Its initial requirements were automating routine tasks, assisting in complex tasks, handling unusual actions, invoking tasks, and supporting status inquiry and explanation. Several shortcomings were revealed, among which the lack of flexibility due to a rigid structure of procedures. *Polymer* and its distributed version *D-Polymer* were developed in the mid-1980s, with an improved focus on goal-based task description. Goal-based cooperation includes activities such as project management, collaborative authoring, large-scale knowledge-base development, and office work [35]. Polymer was focused on goals to be achieved by agents, allowing for ad hoc changes to office work for this end. Slowly, the focus in the industry shifted towards a *tool perspective*, in which the way individuals manipulate artifacts is central.

2.2.3 Business Process Reengineering

The advent of what is called Business Process Reengineering (BPR) was triggered in 1990 by two articles [2]: [24, 14]. In [2], a definition is given for Business Process Reengineering: “a complex, top-down driven and planned organizational change task aiming to achieve radical performance improvements in one or several cross-functional, inter- or intra-organizational business processes whereby IT is deployed to enable the new business process(es)”.

In [24], a case is made for performing total reengineering of business processes instead of merely devising a software tool to assist in the performance of an old way of doing business. They argue for ‘reengineering’ businesses “by using the power of modern information technology to radically redesign our business processes in order to achieve dramatic improvements in their performance”. It is more than just a methodology: “at the heart of reengineering is the notion of discontinuous thinking - of recognizing and breaking away from the outdated rules and fundamental assumptions that underlie operations”.

In [14], the relationship between IT and BPR is seen as a recursive one: thinking about information technology should be about thinking how it supports new or redesigned business processes, and business processes and improvements should be considered in terms of the capabilities information technology can provide. In the article, a five-step approach is designed for performing BPR: develop the business vision and process objectives, identify the processes to be redesigned, understand and measure the existing process, identify IT levers, and design and build a prototype of the new process.

2.2.4 Recent Developments

With the enormously surging popularity of the World Wide Web, changes have come to the workflow industry. In the last decade, the focus has gradually shifted from proprietary workflow solutions to workflow languages that are standardized and used across different organizations.

Following web technology, the architecture of workflow execution has adopted a service-oriented approach in the Business Process Execution Language for Web Services (BPEL4WS). Instead of creating a fully-fledged, monolithic workflow system, BPEL4WS is focused on providing control-flow scheduling of activities that are implemented elsewhere.

In the field of business process notation, the Business Process Modeling Notation (BPMN) has been developed: a graphical language created for being used by both business analysts and information technologists, to bridge the gap between them. Both BPEL4WS and BPMN will be discussed in chapter 9.

Also, the Workflow Management Coalition has devised standards for Workflow Management System architecture: the Workflow Reference Model; and process definition language architecture: the XML Process Definition Language (XPDL). Both are discussed in chapter 9.

2.3 Characterizing Workflows

While workflows can in principle be defined on paper, and serve as a guide to performing a business process, they are normally implemented within the context of an IT system to provide computerised support for the procedural automation of these tasks [27].

Georgakopoulos et al. [22] have written an extensive section on characterizing workflows. They recognize several dimensions along which to describe workflow kinds:

- repetitiveness and predictability of workflows and tasks
- human-controlled vs. automated
- requirements for the systems managing the workflows
- isolation of the system vs. integration of heterogeneous, autonomous or distributed information systems

Among the characterizations of workflows they review is a division into *ad hoc workflows*, *administrative workflows* and *production workflows*, borrowing from McCready [38]:

- *Ad hoc workflows* are essentially human-directed workflows in the context of a small team of professionals working together to reach a certain one-time goal, involving coordination, collaboration or co-decision. In these processes, there is no set pattern for moving information among people. The software supporting this kind of workflows is often called *groupware*, or software for *computer supported cooperative work (CSCW)*.
- *Administrative workflows* are relatively simple workflows involving document routing and authorization processes. The systems directing these workflows have a non-mission critical nature.

- *Production workflows*, altough being repetitive and predictable, are complex and involve multiple information systems. They are mission critical and must deal with integration and interoperability of heterogeneous, autonomous or distributed information systems.

We will mostly look at the latter two workflow types. *Ad hoc workflows* provide a flexibility that is indispensible for certain situations. Still, we focus on the type of software system in which a clear understanding of repetitive processes is used to enable participants to improve the efficiency of these processes.

2.4 Information System Challenges

Workflows are an interesting field of work from the perspective of information systems, as they involve many different elements. Here, we name some of them which, we believe, are the ones that add complexity to an information system. These are the issues that we want to address in WebWorkFlow.

– **Authentication, authorization and administration of participants (users), and roles.** For a workflow system to be able to know who can perform which tasks, a system for authenticating is necessary. Administration for users is also needed. As persons can attain different roles, it is necessary to have some administration facilities for roles and the connection between roles and persons. The currently logged in user should only be able to perform the tasks that are designated to him.

– **Process execution, or task scheduling.** a workflow can be relatively simple, like consecutive execution of a set of tasks, just like performing the consecutive commands in a computer program. But it can also be quite complicated, as in the case where multiple tasks can be done simultaneously, and depending on the outcome of these tasks, the workflow chooses one path or the other. As this involves parallel execution of tasks, it is apparent that the familiar paradigm of linearly executing programs cannot be applied to more complicated workflows. Instead, the paradigm of multi-threaded applications provides a better comparison.

– **Long-term execution.** Another difference between workflows and conventional computer programs, is the fact that workflows must be able to run for a long time. Some workflows take over a year to finish, and a blocking program waiting for a year to finish is not an ideal situation. Instead, workflows must be able to be saved and resume execution later.

– **Viewing and performing tasks.** Tasks need to be performed. This can be done by a participant, or - in the case of an automated task - by a computer. After the task has been performed, control must be given to the workflow task scheduler. In case of a manual task, the task should be viewed somehow. Therefore, a user interface should be provided for this cause.

2.5 Examples of Workflows

Virtually every activity that is performed repetitively can be modeled using a workflow. In this section, we will give some examples of procedures in which workflows can play a benificiary role.

2.5.1 Personal Development Plan

Consider the following case: a company has an employee review procedure that is done twice a year, during which the personal development plan of the employee is evaluated and adapted to current events. The procedure involves a conversation between an employee and his manager, and procedes as follows:

1. Both the employee and his manager fill in a preparation form for the conversation.
2. The conversation takes place, during which the manager enters a report of the conversation.
3. After the report has been written and the conversation is finished, the employee reads it, and if he agrees that the report describes the contents of the conversation well, he approves the report and the workflow ends. However, he can also choose to disapprove by making comments on the report. After this, the report must again be changed by the manager, after which the employee has a new chance to agree on the contents of the report.

This workflow involves multiple actors and several documents that have to be edited during the process. The possible next steps depend on the tasks that have already been performed during the process.

The PDP workflow is a relatively simple, repetitive process that is performed for every employee. More complex workflows are possible as well.

2.5.2 Academic Conference

Consider the process of organizing an academic conference. This is a process that stretches over more than a year, involving many actors and documents, and is considerably more complex than the previous example. As an example, take ACM/SIG's conference organization procedure [11]. Roughly, it consists of the following steps:

1. Submitting a Preliminary Approval Form (at least 24 months before conference), followed by an approval step.
2. Submitting Technical Meeting Request Forms (TMRF) (at least 12 months before conference).
3. Submitting a Mini-TMRF for each sponsored workshop held in conjunction with the conference.
4. Submitting a Joint Sponsorship Agreement Form.
5. Putting together an organizing committee by inviting members, who respond to these requests.
6. Publishing a Call for Papers, with a deadline.
7. Receiving papers.
8. Optionally extending the deadline.

9. Closing Submissions.
10. Enable bidding for papers to review by reviewers.
11. Assigning papers to reviewers.
12. Composing a program.

This is only a very basic view on organizing a conference, in which arranging a venue and guest accommodation is not even included. Also, the steps that are mentioned are very coarse grained.

A formal description of a workflow as complicated as organizing an academic conference can easily become large and complex. The language provided for describing workflows has an important role in deminishing complexity by providing adequate constructs.

2.6 Domain-Specific Languages

Opposed to general purpose languages, *domain-specific languages* are for use in a limited domain. These domains range from very narrow, such as the domain of context-free grammars, to broader, such as the domain of all web pages. An example DSL of the former kind is BNF (Backus-Naur Formalism), a formalism for expressing context-free grammars. Examples of the latter are WebDSL and WebML. According to Mernik et al., some consider Cobol to be a DSL for business applications [41]. If included in the notion of DSLs, its domain would be very broad. A sound definition of a domain is difficult. The closest we can get to a definition may be a circular definition that a domain is “a coherent area of (software) knowledge that can be captured in a DSL” [70].

According to Mernik et al., “DSLs trade *generality* for *expressiveness* in a limited domain.” [41]. *Expressiveness* relates to how adequate the abstractions in a language are for use in the domain they are meant for. Measuring expressiveness could be done by comparing lines of code of a program written in some DSL to an equivalent program written in a general purpose language. If the DSL program is much smaller than the general purpose language program, the DSL is said to be very expressive. In effect, a lot of what is called *boilerplate code* [70], code that is common for the domain intended and must be typed over and over again, is scrapped.

Reasons to create a DSL, other than increased expressiveness, are analysis, verification and optimization. As patterns that in general purpose languages are normally repetitively expressed in low-level abstractions are encoded into abstractions in the DSL, analysis of semantics is much easier. As semantic errors in programs for general purpose languages - which are hard to detect - are sometimes equivalent to syntax errors in DSLs, errors are discovered much sooner. As the implementation of abstractions in DSLs can be tailored for the domain, it is possible to provide a mapping that incorporates optimization compared to general purpose language programs.

Van Deursen et al. define a DSL as follows.

> A *domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. [66]

DSLs, however, are not necessarily executable, while the definition seems to implicate this. In Visser's definition this implication is not made:

> A domain-specific language (DSL) is a high-level software implementation language that supports concepts and abstractions that are related to a particular (application) domain [70].

To get a better idea of what DSLs are, here are a few familiar examples.

- *YACC*, or *Yet Another Compiler Compiler* is a DSL for specifying grammars to generate a parser. It accepts LALR(1) grammars with disambiguating rules, from which a parser implemented in C is generated [31].
- *(La)TeX*, the famous language for typesetting scientific paper, can be seen as a domain-specific markup language.
- *SQL*, or *Structured Query Language*, is considered a DSL as well. It's domain is the domain of querying data from relational databases.
- *Excel* can be viewed as a DSL for scientific computations, while in fact the domain is even bigger than just scientific computations.

As clearly, a lot of different domains and types of DSLs exist, we will now discuss some different types of DSLs. We will discuss *application frameworks*, *internal languages*, *embedded DSLs* and *compiled DSLs*, all mentioned by Visser in [69].

2.6.1 Application Frameworks

A straightforward way to prepare domain-specific knowledge for reuse in applications, is building an *application framework*. Being a library, one could argue about whether it really is a separate language.

Mernik et al. compare application frameworks to proper DSLs in [41]. In their opinion, DSLs have some advantages over application frameworks:

- DSLs offer appriopriate domain-specific notations, which are directly related to the productivity improvement associated with the use of DSLs
- Functions and objects often do not suffice for replacing appropriate domain-specific notations, due to incompatibility of certain constructs such as traversals or error handling.
- DSLs provide more possibilities for analysis, verification, optimization, parallelization, and transformation.

However, deciding to develop a DSL instead of an application framework is not always a good idea, as there are some disadvantages as well:

- DSL development is hard, requiring both domain knowledge and expertise in the field of language development.
- Estimating the return on value of DSLs is hard, as one never knows the outcome of a DSL design project at the start.

Therefore, before deciding to develop a DSL, one should carefully balance the facts. Only if the language will be used for many applications, developing a DSL is worth the trouble. Otherwise, putting effort in developing an application framework may just be the right thing to do.

2.6.2 Internal Language

Some languages, such as LISP [37], provide possibilities for extension. This is, in fact, a way to syntactically incorporate the use of a library in the language. Giving slighly better ways of adapting the language, this kind of language extension can be thought of as a different class of DSLs from application frameworks.

2.6.3 Embedded DSL

A DSL can also be used to provide constructs to express certain abstractions in an adequate way, while not having to change a whole language. This can be done by providing a language embedded in another language, to which it is first transformed. In effect, this can be thought of as adding syntactic sugar, without changing a language's features. An embedded DSL is more than a way to transform strings of code into the host language, as its syntax is checked during the process, including elements of the host language (defined variables) that are used in the guest language code.

An example of this is the language SWUL (SWing User-interface Language), an embedded DSL for Swing user interfaces for use within Java code [9]. When compiled, any program fragments written in SWUL will first be transformed to Java code, after which the Java code can be compiled separately. The benefit of the language is an easy way of specifying user interfaces in Java, instead of the tedious old way of defining nested user interface objects in which encapsulation of objects in other objects was difficult to express concisely.

Another example of an embedded language is LINQ (Language-Integrated Query) [40], a language for specifying SQL queries inside .NET code in which variables of the host language can be used, while avoiding the danger of SQL injection.

2.6.4 Interpreted DSL

An interpreted DSL is a language that is not compiled, but parsed and interpreted on the fly. An example of an interpreted DSL is *SQL*. When executing an SQL query, the SQL command is first parsed and then immediately executed.

2.6.5 Compiled DSL

A compiled DSL, also called *external DSL*, is a type of DSL from which executable code is generated using a compiler. Examples of compiled DSLs are WebDSL and WebML. The model is specified using the DSL, after which a compiler transforms the model into a working application in some lower-level language, usually a 3d Generation Programming Language (3GL), whereas the DSL can be thought of as a 4th Generation Programming Language (4GL).

2.7 Domain-Specific Language Design Methodologies

In the field of domain-specific language design, several methodologies can be helpful. We will discuss both a *deductive* and an *inductive* approach to modeling domain abstractions. After that, we will discuss *generation by model transformation* and the benefits of the use of a *core language*.

2.7.1 Deductive Approach

In [66], the following design methodology for DSLs is proposed: identify the problem domain, gather, all relevant knowledge in the domain, cluster this knowledge in a handful of semantic notions and operations on them. Design a DSL that concisely describes applications in the domain. Construct a library that implements the semantic notions and design and implement a compiler that translates DSL programs to a sequence of library calls. Using this approach, which could be called *top-down* or *deductive*, the domain as a whole is viewed first. After this, the semantic notions and operations are derived in one single step, preceding design and implementation.

2.7.2 Inductive Approach

Visser introduces a different approach in [70]. He argues that designing a DSL should be done iteratively, to minimize the risk of failure using a gradual development. Moreover, it should be done in an *inductive*, technology-driven way, because this leads to faster results based on firm grounds. However, one has to be careful that the technology-driven approach does not lead to *leaky abstractions*, in which the DSL is too dependant on the underlying target technology.

While a deductive approach has as an advantage that it usually leads to a coherent and well-thought of domain analysis, an inductive approach usually leads to faster results, as the translation to the underlying technology is much more straightforward.

2.7.3 Generation by Model Transformation

In [25], Visser et al. introduce the method of *Code Generation by Model Transformation*. Using Stratego [8], a toolset for model-driven development using *rewrite rules* combined with programmable *rewriting strategies*, they construct a code generating compiler.

Stratego uses SDF [68], a syntax formalism to define a grammar, from which a scannerless parser is generated. Feeding a model into this, results in an abstract syntax tree in the ATerm format, a concise format for representing trees. This AST is subsequently rewritten using rewrite rules and strategies, resulting in the generation of executable target code. In this process, the AST is a representation of the model. The input model is transformed by applying rewrite rules and strategies, resulting in a transformed model. Only at the end, text-based code is generated from the model.

This approach results in a compiler that exhibits separation of concerns in multiple dimensions. *Vertical modularity* is accomplished by dividing the process into diffent stages, consisting of separate transformations that are applied sequentially, each concerned with their own aspect of the code generator. *Horizontal modularity* is accomplished by being able to define separate rules for transformation that are applied at the same stage. This allows for extending the generator meta-model with new abstractions.

2.7.4 Core Language

Exploiting the vertical modularity, a language can be built in different layers of abstractions, stacked to make up a complete compiler. A central role in this sequence is fulfilled by a *core language* [70], a set of the most basic constructs of the input language. During compilation, there is a stage in which the model consists of only these basic constructs, before starting the process of generating code from the model. The part of the compiler transforming the input model to the basic core language level, and the part of the compiler transforming the core language model to the target code can respectively be referenced as *front-end* and *back-end* of the compiler.

This approach has several benefits. A well-designed core language can be an incentive to organize the compiler code well. Moreover, having a separate front-end and back-end enables the straightforward extensibility of the compiler to different target environments. Complicated transformations in the front-end compiler only have to be thought of once. When implementing a new back-end for the compiler, only a transformation from the relatively simple core language to the target environment needs to be developed.

Chapter 3

WebDSL

Being the basis on which WebWorkFlow is built, WebDSL is explained in this chapter. The WebDSL project was started by Eelco Visser as a case study in domain-specific language engineering [70]. Since, it has evolved from a language containing basic constructs for web pages, to a language with an advanced templating system, an access control sub language [23], and advanced styling mechanisms.

In this chapter, we will first discuss WebDSL's compiler architecture, after which the different elements of the language are explained. We will introduce entities, pages, and present action code. We will introduce templates as a method of modularizing WebDSL user interface code. We will explain the different user interface elements and control-flow elements, and we will finish by demonstrating the access control language.

3.1 Compiler Architecture

The WebDSL compiler is implemented using Stratego/XT, a toolset for program transformation [8]. This toolset provides specification of a grammar using Syntax Definition Formalism (SDF) [68], that allows for automatic generation of a scannerless parser, accepting any context-free grammar. It uses ATerms [6, 7], or Annotated Terms, for representing the resulting Abstract Syntax Tree (AST). This AST is subsequently transformed using Stratego, a language for specifying transformations using rules and strategies. These strategies consist of the application of rules and other strategies. After transformation, the resulting program representation - still consisting of ATerms - is converted into code and written to files.

The AST transformation process allows for vertical and horizontal modularization, which makes for perfect conditions to extend the compiler with another transformation step. The WebDSL compiler is organized as a stack of transformations that are performed consecutively. These transformations are splitted into two general processes: the front-end and the back-end of the compiler. The front-end of the compiler converts the input into what is called *Core WebDSL*. From here, the back-end generates a fully functional application for a specified platform. This architecture makes it possible to add a new target technology by only adding a new back-end part of the compiler. This

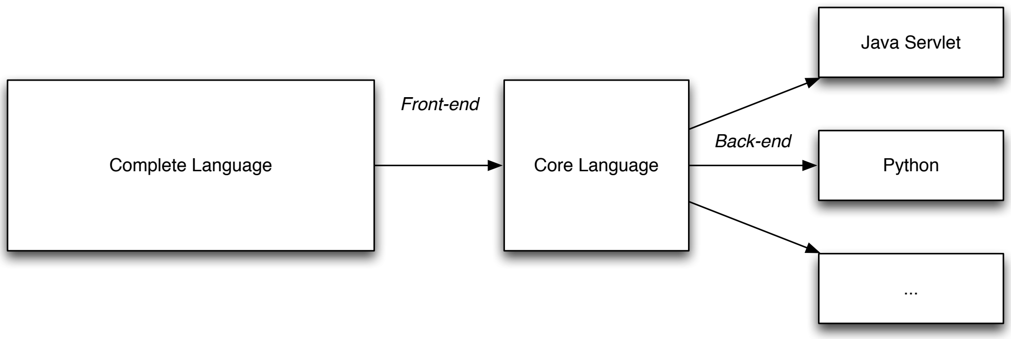


Figure 3.1: Compilation using an intermediate Core Language

means the transformation of the core language to the target technology. At the moment, WebDSL has two working back-ends, namely *Java Servlets* and *Python*.

3.2 Entities

To define a data model, WebDSL provides *entities*. The following example demonstrates some aspects of defining entities:

```
entity Book {
  title    :: String (name)
  location :: String
  pages    :: Int
  author   -> Author
}

entity Author {
  name  :: String
  books -> Set<Book> (inverse=Book.author)
}
```

Attributes are defined using the syntax `Name PropKind Type (Annotation*)`. The annotation part is optional, and can be used to give extra information about the property. A PropKind can be either `::` (simple), `->` (reference) or `<>` (composition). A simple propkind is a built-in type like String or Int. A reference type is either a defined entity type or a set or list of defined entity types. A composition signifies that when one of the two entities is deleted, the other should be deleted as well.

In this example, two separate entities are defined. Books consist of two strings and an integer. The title of a book is the attribute that is to be used as its name, signified by the `(name)` annotation. WebDSL uses this information to automatically display this attribute when referring to an object of this type. A book is also written by an author, which is a many-to-one relation. This is modelled by books having a reference to an author, and authors having a reference to a set of books. At the the author side, some extra information is given using annotations: which attribute of Book is the one that signifies the other side of the relation. This information makes it a two-way connection instead of two one-way connections.

3.2.1 Subtyping

An entity can be defined as a subentity of an existing entity using inheritance, with the following syntax:

```
entity NonFictionalBook : Book {
  yearOfEvents :: Date
}
```

This will create an entity type NonFictionalBook with all properties of Book, extended with an attribute yearOfEvents.

3.2.2 Derived Attributes

Besides normal attributes, entities have derived attributes. A derived attribute is an attribute that cannot be set, but is calculated from the value of other attributes. An example is given in the following code snippet:

```
extend entity Author {
  bookCount : Int := this.books.length
}
```

The `bookCount` attribute derives its value from another attribute books, which is accessed by prefixing it with `this.`. Extending an entity in this case has nothing to do with inheritance, but is simply a way to add attributes to an existing entity, at a different location within the code.

3.2.3 Entity Functions

Entities can also contain functions. These functions are called entity functions.

```
extend entity User {
  function canEditBook(book:Book) : Bool {
    if (this.isAdmin) { return true; }
    for (a : Author where a.name == this.name && a == book.author) {
      return true;
    }
    return false;
  }
}
```

In this example, the function `canEditBook(book:Book)` is added to entity User. It first checks if the user is an administrator, in which case `true` is returned. Then it checks if the user has authored the book. If so, he can edit the book. Otherwise, he cannot, and `false` is returned.

3.3 Pages

In WebDSL, pages are a central abstraction, as they are usually the central elements in web applications. In WebDSL, a page can be defined in the following way:

```
define page allBook() {
  header { "Book Index" }
  section {
    table {
      for (book : Book) {
        row {
          navigate(viewBook(book)){text(book.name)}
          navigate(editBook(book)){"edit"}
          navigate(deleteBook(book)){"delete"}
        }
      }
    }
  }
}
```

This example shows the use of headers, sections, tables, and hyperlinks.

Pages can be referenced like functions can be in most languages. A page reference can also contain arguments, as is shown in the example links to `viewBook(book)`, `editBook(book)` and `deleteBook(book)`.

The for-loop `for (book : Book) {}` iterates over all existing entities of the type Book, using the items in the collection of Book as input for the `navigate` elements.

A navigate element `navigate(pageName(arguments)){text}` creates a hyperlink to the page identified by `pageName`, displaying what is entered between the curly braces.

3.4 Action Code

While pages are mostly declarative, as they merely describe the contents of a page, WebDSL does also have imperative language parts. The imperative sublanguage of WebDSL is called *action code*. Actions can be defined on pages, triggered by a button. Often, their context is a form on a page, as shown in the next example:

```
define page newBook() {
  var newBook : Book := Book{};
  header { "New Book" }
  section {
    form {
      table {
        row { text("Name: ") input(newBook.title) }
        row { text("Location: ") input(newBook.location) }
        row { text("Amount: ") input(newBook.amount) }
        row { action("Add book", add()) }
      }
    }
    action add() {
      if (newBook.title != "") {
        newBook.persist();
        return message("Book " + newBook.title + " succesfully added.");
      } else {
        return error("Title cannot be empty.");
      }
    }
  }
}
```

This example demonstrates a page definition for adding an entity of type `ExampleEntity`. In this example, a new variable of type `ExampleEntity` is declared at the start. Inputs are created for the attributes of this new entity, placed on a form that has an action `add()` attached to it. When viewing the page, at the place of `action("Add entity", add())`, a button is displayed that triggers the action.

The input construct automatically causes the page to display a suitable form input for the data type of its argument.

The action consists of a test whether the user has entered a value for name. If the test succeeds, a new entity is added and the user is redirected to a page called `message`, that takes a string as an argument. Otherwise, an error is displayed by redirecting the user to a page `error`. Redirecting the user to a page is done by returning the page at the and of an action definition.

Saving the entity is done by calling the function `persist()` on it. This causes the application to save its contents in the database, available for later retrieval.

Besides in actions, action code can be used in `init{}` sections, code sections that are executed when the user views the page. In an init section, `return` cannot be used. Instead, `goto` can be used to redirect a user to a page.

3.5 Templates

WebDSL has a template system to encapsulate parts of pages for reusability. The template engine is part of the compiler, but in such a way that the compiler could also work without the template part. This exploits the possibility for vertical and horizontal modularization. The template engine is not merely executed as a step before the rest of the WebDSL compilation, but is merged into the compiler using the possibilities for horizontal modularization. Nonetheless, in the picture of the different layers of abstraction (figure 3.2) they are viewed as separate layers.

While the template engine adds considerable usability to the WebDSL language, it does not change anything to the language possibilities, as all transformations make use of the underlying Core WebDSL.

The template system allows to define, redefine or call a template. This is illustrated in the following example.

```
define main(page : String) {
  navigation(page)
  body()
}
```

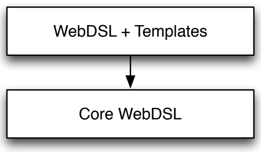


Figure 3.2: WebDSL including Templates

```
define navigation(page : String) {
  if (page != "Home")    { navigate(home()){"Home"}       }
  if (page != "Contact") { navigate(contact()){"Contact"} }
}
define body() {}
define page home() {
  main("Home")
  define body() {
    section("Home") {
      "Welcome to this web site"
    }
  }
}
```

In this example, the template `main()` is defined for use in a general web page. It contains both a `navigation` part and a `body` part, which will be used for the actual page content. The navigation template consists of navigation links, of which only the pages that are not currently displayed are visible. The `body()` template is defined as an empty template.

The use of `main()` is illustrated in the definition of page `home()`. This page calls `main()` but overwrites the empty `body()` template with some actual content. Instead of displaying the formerly defined `body` template, `main()` will display the newly defined `body` template, as it is defined within the same scope.

This example also illustrates the use of arguments to templates. In this case, `main` is called with an argument: `main("Home")`. This argument is passed on to `navigation`, where it is used to determine whether the link to the home page must be displayed or not.

3.6 User Interface Elements

In this section, we explain the different UI elements that can be used within pages and templates. UI elements are separated by any whitespace; not differentiating between spaces and line breaks. Whitespace is not printed.

3.6.1 Text

Simple text is printed by placing it between double quotes. For instance:

```
define page greet(name : String) {
  "Welcome to this page, " output(name)
}
```

...will simply print the text between double quotes, augmented with a name provided. This name is printed using a call to output, which will automatically choose a suitable way of printing the contents of the variable provided. In this case, its execution will be equivalent to using `outputText()`.

Other functions for outputting text are *outputBool*, *outputString* (which prints a text field), *outputInt*, *url*, *outputFile* (which prints a download link), *outputImage*, *outputWikiText*. The following example illustrates the use of different functions for printing the same variable contents.

```
define page redirect(link : Url) {
  "This is a link to " outputText(link) ": " outputUrl(link){"Go!"}
}
```

This page will first print the given URL, and then a link with the text ‘Go’, referring to the given URL.

Input functions work similarly, and have similar names like *inputBool* and *inputText*. All mentioned output functions have their input counterparts. Added to this is a *captcha* element for use in forms to make sure that the person filling in the form is, in fact, a human being.

3.6.2 Basic Elements

- `title{"New page"}` sets the page title to ‘New page’.
- `header{"Heading"}` will generate a text header, adjusted for the current depth. If used inside a section, a *h2* will result instead of a *h1*.
- `spacer` or `horizontalspacer` will produce an HTML *hr* element.
- `image("/img/logo.png")[height:=100px, width:=20%]` will result in an image being displayed.
- `pre{"Preformatted text"}` can be used for literally showing the contents, maintaining whitespace.
- `label(string)` is used to render an HTML *label*. If used in a group or row element, any children will be placed in different columns. This makes it easy to nicely format an input form.

3.6.3 Tables

A table can be constructed using the following syntax:

```
table {
  row {
    column{"Text on the left"} column{"Contents on the right"}
  }
  row {
    column{"Text, too"} column{"And yet more contents"}
  }
}
```

If a column contains but a single element, however, the column signifier can be left out.

3.6.4 Container Elements

- `block("blockStyle", "12"){ ... }` is used to place contents inside an HTML *div* section. The first argument to block will result in the application of a CSS style. The second argument can be used to attach an id to the *div*.
- `section{ ... }` can be used to create inlined sections of content, taking track of depth to automatically adjust font size.
- `container{ ... }` is equivalent to sections, except for the nesting depth effect.
- `par{ ... }` creates a paragraph.
- To create a list, the following syntax can be used:

```
list {
  listitem { ... }
  listitem { ... }
}
```

- Similarly, elements can be grouped. A grouped set of elements is rendered with borders and with a caption displayed in bold text.

```
group("Caption") {
  groupitem { ... }
  groupitem { ... }
}
```

3.6.5 Navigation Elements

To create links between different pages in WebDSL, a *navigation element* is used. There are two different navigation elements: `navigate(target){ ... }` and `navigatebutton(target, string)`. The former renders the content as an HTML link to *target*. The latter creates a button for going to the targeted page, with a string to give the button a caption.

Targets can be constructed using the page name and optional arguments between brackets. For instance, `navigation(displayAuthor(author)){ "View author" }` will print the text ‘View author’, with a link to page *displayAuthor*, using *author* as a parameter.

3.6.6 Menus

To create a menu, the following example code can be used.

```
menubar("horizontal") {
  menu {
    menuheader{"Authors"}
    for (a : Author) {
      menuitem { output(a) }
    }
  }
  menu {
    menuheader{"Contact"}
    menuitem{ navigate(sendEmail()){"Email"} }
  }
}
```

A *menubar* element is used to start either a horizontal or a vertical menu. Inside the menubar, *menu* elements are used. They should contain at least a *menuheader* and possibly one or more *menuitems*.

In this example, two menus are created. The first is a menu of all authors, displaying links to the viewpages for authors (which is automatically done by using *output* with the author as an argument). The second is a small menu with only a static link to a page *sendEmail*.

3.7 Control-Flow Elements

To influence the flow of rendering within a page or template, several control-flow elements are provided. These elements can also be used as statements in action code.

3.7.1 Variable Declarations

Declaring a variable is done in this way:

```
var newVariable1 : String;
var newVariable2 : Int := 3;
var newAuthor : Author := Author {
  name := "Lewis, C.S."
}
```

The first example creates a *null* string variable. The second example creates an integer that is initialized to 3. The last example creates a variable of a custom type *Author*, and initializes it to a newly created author.

3.7.2 Conditional Code

As in most programming languages, an *if-then-else* construct can be used for executing conditional code or conditionally displaying certain elements. The *else* clause is optional. Use of curly braces, however, is mandatory. An example:

```
if (author.lastName == "Lewis") {
  "C.S. Lewis"
} else {
  "Other writer"
}
```

3.7.3 Loops

WebDSL provides a for-loop for iteration. This is a powerful statement that is able to automatically fetch objects from the datastore to iterate over them. It also accepts a *filter*, *order expression*, *limit* and *offset*, as is used in `SQL SELECT` statements. We will now give a few examples of the use of `for`.

```
for (a : Author) {
  navigate(a)
}
```

This example will simply print navigate links to all authors in the datastore.

```
for (a : Author in authorList where lastName == "Lewis" order by firstName desc) {
  navigate(a)
} separated-by { ", " }
```

This will print navigate links to all authors in the list *authorList* with ‘Lewis’ as a last name, reverse ordered by first name. The `separated-by` constructs adds a comma in between the links.

3.8 Access Control Language

On top of the template engine, WebDSL is extended with a sublanguage for access control, to provide authentication and security to web pages. This language, again, does not provide any new possibilities to the applications that can be generated, but merely adds functionality by improving usability. The abstraction stack including the Access Control Sublanguage is displayed in figure 3.3.

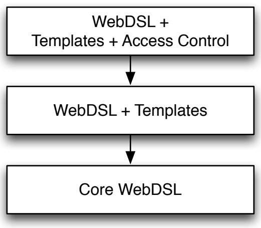


Figure 3.3: WebDSL extended with Templates and Access Control

3.8.1 Configuring Access Control

WebDSL access control makes use of a session entity to hold the state of the presently logged in user. This session entity is configured by:

```
access control rules {
  principal is User with credentials name, password
}
```

This code says that the User entity should be used to represent logged in users, and that its credentials can be found in the attributes `name` and `password`.

3.8.2 Authentication

After configuring, authentication can be done as shown in this code:

```
...
action login() {
  for (u : User where u.username == username) {
    if (u.password.check(password)) {
      securityContext.principal := u;
      securityContext.loggedIn := true;
      return home();
    }
  }
  securityContext.loggedIn := false;
  return error("Wrong combination of username and password");
}
```

By assigning the user that is logging in to `securityContext.principal`, and setting the attribute `loggedIn` to `true`, its authentication is saved in session context.

3.8.3 Protecting Resources

Using access control, three levels of can be used: page, template and action. Moreover, there are several constructs that provide syntax for shorter specifications, notably *pointcuts*, which are groups of resources that can be referred to afterwards by using the pointcut name, and *predicates*, a way to encapsulate complex boolean expressions to allow for reuse. We will only demonstrate securing a page here. For more detail into the access control language, the reader is referred to [23].

Securing a WebDSL page is done by providing conditions for someone to be able to view the page:

```
rule page editBook(b) {
  securityContext.loggedIn &&
  securityContext.principal.canEditBook(b)
}
rule page editAuthor(*) {
  securityContext.loggedIn &&
  securityContext.principal.isAdmin()
}
```

This example shows that rules are used to secure pages, and that conditions are specified in a declarative manner. However, by using a predefined function `canEditBook(b)`, imperative code can be used to aid in the process of determining whether the user is allowed to view the page. If however, a user should go to the page and not have access, an “Access denied” message is displayed.

3.8.4 Hiding Protected Resources

The access control restrictions will make pages inaccessible to some users. Displaying navigation links to pages that, after requesting them, turn out to be inaccessible, is not a very elegant way of securing resources. Therefore, when a page is not accessible to a user, any navigation links to them will not be displayed. Also, outputting objects that would normally render a link to a page that is now prohibited, will result in printing the name only.

Chapter 4

WebWorkFlow

WebWorkFlow is our answer to the need for an integrated workflow application language. Using WebWorkFlow, a web-based workflow application can be built using one, coherent specification of the process and all procedures it consists of.

WebWorkFlow is a textual language. As such, it is very close to regular programming languages, providing constructs like functions and variables, as most programming languages do.

WebWorkFlow provides a highly expressive *process language* in which most patterns used in workflow applications can be easily expressed. If, however, it is not possible to specify a certain control-flow pattern using the process language, WebWorkFlow enables developers to resort to the abstraction level just below processes: *procedures*. These procedures are the basic blocks of process-oriented web applications. They can be either an atomic procedure or a procedure containing a process description. When process descriptions in which lower-level procedures are brought together do not suffice, it is possible to amend the lower-level procedures in such a way as necessary to further customize a process.

WebDSL is built using a bottom-up approach. The first layer (Core WebDSL) is a relatively thin layer on top of the abstractions available in the target platforms. The other functional layers such as templates and access control, are added as transformations to the next layer. Being built in this way, WebDSL provides adequate abstractions for artefacts like pages and functions, which are available in the target platforms, as well. An important difference is that the WebDSL abstractions are much more concise and form a language that is extensively typechecked. However, WebWorkFlow is built to provide process-oriented abstractions, instead of abstractions for web pages. This adds abstractions for an altogether different paradigm. As every WebWorkFlow construct is translated by the compiler to access control level WebDSL, all applications constructed using WebWorkFlow can in principle be expressed in WebDSL, but this is a tedious process in which mistakes can easily be made. Using WebWorkFlow, process-oriented applications can be constructed in a much more concise way.

The result of a web application specification in WebWorkFlow is a web application consisting of pages that are connected to form a process. WebWorkFlow makes heavy use of the access control language in WebDSL, to make sure that tasks in a process can only be performed by the actors that are supposed to perform them, in the correct order.

In the rest of this chapter, we will demonstrate the use of WebWorkFlow. First, we will give an example of the use of the *process language*. In the same section, we will elaborate on the basic building blocks used by process-level workflow descriptions: *procedures*. The section also includes an implementation and a demonstration of the example workflow. Finally, we will give an example of using the procedure-level abstractions for defining a workflow instead of a process definition, to achieve more flexibility than can be reached using just process-level abstractions.

4.1 Process Level: Expressiveness

In this section, we will demonstrate the different elements used in defining a workflow: an expressive *process description* and a definition of several *procedures* that are used in the process description. We will also demonstrate the application that is specified in this section. We will use the example of the *Personal Development Plan (PDP)* workflow, given in section 2.5.1.

4.1.1 Process Description

A process description is a relatively short expression describing the workflow from the control-flow perspective. In the process description, the control-flow between separately defined *procedures* is specified. In this description, one can use several control-flow abstractions, like sequential composition, conditional execution, parallel execution and loops.

We will give an example of the *PDP* workflow of section 2.5.1, after which we will explain the different elements that comprise the process description.

```
auto procedure pdpWorkflow(p : PdpMeeting) {
  process {
    (employeeFillInForm(p) and managerFillInForm(p));
    while (!p.approved) {
      writeReport(p);
      approveReport(p) + commentReport(p)
    }
  }
}
```

The constructs that look like function calls are references to procedures, passing an argument to signify on which object the procedure is supposed to run.

This example illustrates several different control-flow constructs:

- *parallel execution:* the expression `employeeFillInForm(p) and managerFillInForm(p)` is treated as a parallel execution of both filling in the *employee preparation form* and the *manager preparation form*. After completing one of the two procedures, the workflow waits for the other branch to complete, before carrying on with its execution.
- *sequential composition:* the semicolon schedules two procedures to be executed in succession.
- *while:* the while construction causes the contents of its associated block to loop until the reentry-condition becomes false; in this case, when the report is approved. The manager first

writes the conversation report. If the employee agrees with the description of events as given by the manager, he approves the report. If, however, the employee thinks the report contains errors, he leaves his comments on the report. The `while` causes `writeReport` to become available again, after which the employee has a new chance of approving the report.

- *deferred choice (+):* the deferred choice construction enables two parallel branches. However, only one of the branches is executed in the end. The choice between the branches is made by performing an activity in one of the branches. At the moment an activity is performed, the branch in which the activity resides is chosen, and the other branch is disabled.

The rest of the available control-flow constructs will be explained in chapter 6.

4.1.2 Procedures

While the process description is a view on the workflow from a control-flow perspective, the procedures referenced in process descriptions can be seen as a view on the workflow from a resource perspective.

Procedures are the basic building block of process-oriented web applications. They encapsulate a certain activity or task. They may or may not contain a *view page*, a *specified actor or group of actors*, *conditions* for making the procedure available, an *action* and a *process description*. These are specified by respectively `view {}`, `who {}`, `when {}`, `do {}` and `process {}`.

The procedures that are used in the *PDP* workflow are displayed here.

```
procedure employeeFillInForm(p : PdpMeeting) {
  who  { securityContext.principal == p.employee }
  view { derive procedurePage from p for (employeePreparation) {
         title{"Fill in employee form"}
         header{"Fill in employee form"} } } }

procedure managerFillInForm(p : PdpMeeting) {
  who  { securityContext.principal == p.employee.manager }
  view { title{"Fill in manager form"}
         derive procedurePage from p for (managerPreparation) {
           title{"Fill in manager form"}
           header{"Fill in manager form"} } } }

procedure writeReport(p : PdpMeeting) {
  who  { securityContext.principal == p.employee.manager }
  view { derive procedurePage from p for (report) {
           title{"Write report"}
           header{"Write report"} } } }

procedure approveReport(p : PdpMeeting) {
  who  { securityContext.principal == p.employee }
  do   { p.approved := true;
         p.persist(); } }

procedure commentReport(p : PdpMeeting) {
  who  { securityContext.principal == p.employee }
  view { derive procedurePage from p for (comments) {
         title{"Comment report"}
         header{"Comment report"} } } }
```

In this example, two procedure styles are displayed. The procedures `employeeFillInForm`, `managerFillInForm`, `writeReport`, and `commentReport` contain an actor, specified by `who` {}, and a view definition. This results in a page that is visible only for the specified actor (in the case of `managerFillInForm`: the manager of the employee who is having his PDP meeting), containing fields for editing the value of `p.report`. The procedure `approveReport` contains no view definition. As such, instead of a link to a page, a button is visible to perform the procedure activity. `approveReport` also has an associated *action*, signified by `do`. In this case, the value of approved is set to `true` upon performing the procedure. Neither of the procedures have a condition besides the actor. Note that although the procedures with view definitions do not have an explicit action, their view page is generated using a `derive` construct. When using a `derive` construct for a `procedurePage`, an action is generated automatically, persisting the altered associated object after submitting the generated edit form. A `title` and `header` can be given when deriving a page.

Before a procedure can be performed, it has to be *prepared* and *enabled*. Preparing a procedure means that the necessary data elements are created. A procedure is disabled initially. It has to be enabled before it can be performed. Instead of first preparing and subsequently enabling a procedure, the procedure can be *started*, which will do the same. More on this is explained in section 5.3.3. When a process description is specified for procedures, this process description takes care of starting the procedures that are referenced within the process. In our example, the workflow is started after creation of the associated object:

```
var p : PdpMeeting := PdpMeeting{ employee := aUser };
p.persist();
p.startPdpWorkflow();
```

4.1.3 Demonstration

In this section, we will give a short demonstration of the workflow application. We begin by showing you the *all tasks* page, on which all tasks for the currently logged in user are displayed, grouped by the object for which the procedures are defined. In this case, the employee is logged in and the only available task is filling in his preparation form. As can be seen in figure 4.1, this task belongs to the object called ‘Joe User PDP Form’.

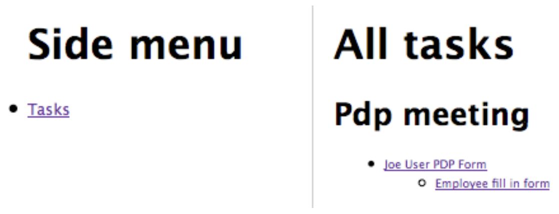
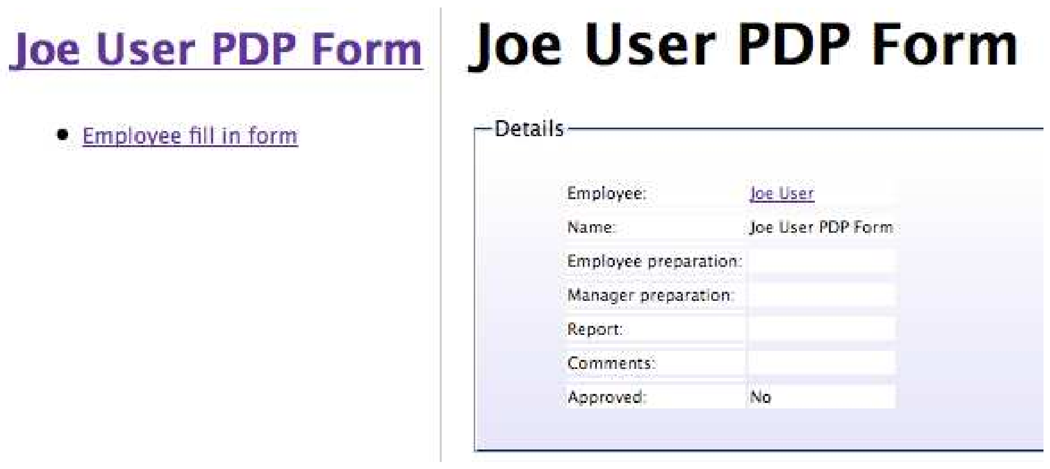


Figure 4.1: All Tasks Page

The remainder of the workflow application is shown in figures 4.2 through 4.4



Joe User PDP Form

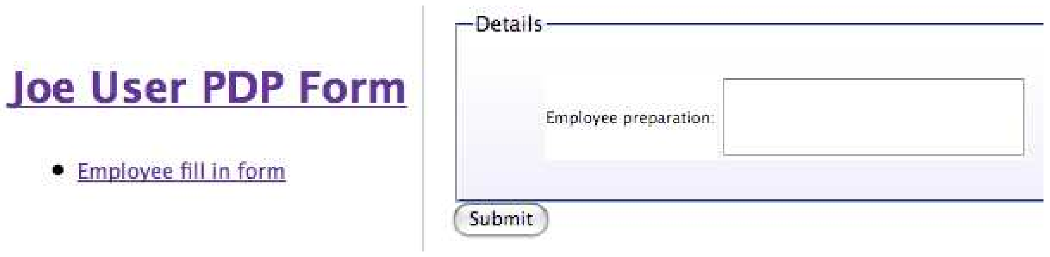
- Employee fill in form

Joe User PDP Form

Details

Employee:	Joe User
Name:	Joe User PDP Form
Employee preparation:	
Manager preparation:	
Report:	
Comments:	
Approved:	No

In this screenshot, the view page for 'Joe User PDP Form' can be seen. When viewing an object, all relevant available procedures are displayed in the sidebar on the left. In this case, a link to the only available procedure is displayed. After clicking this link, the following screen results.



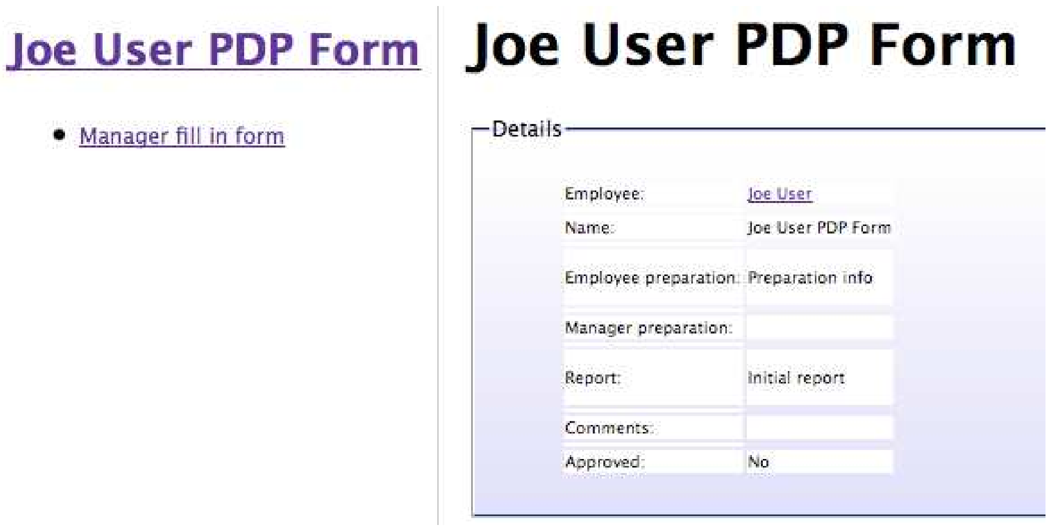
Joe User PDP Form

- Employee fill in form

Details

Employee preparation:	

Filling in the preparation form and submitting it, directs the user back to the object view. After noticing that no more procedures are available, we log in as the manager, and see that there is a procedure available for execution by the manager.



Joe User PDP Form

- Manager fill in form

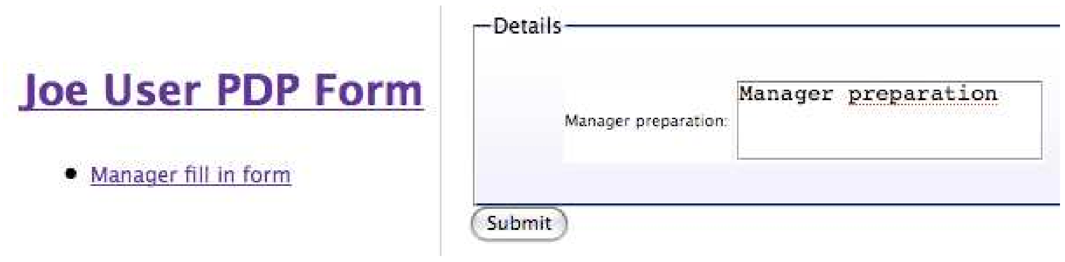
Joe User PDP Form

Details

Employee:	Joe User
Name:	Joe User PDP Form
Employee preparation:	Preparation info
Manager preparation:	
Report:	Initial report
Comments:	
Approved:	No

Figure 4.2: Workflow Demonstration

Analogous to filling in the employee form, filling in the manager form leads to the next screen:

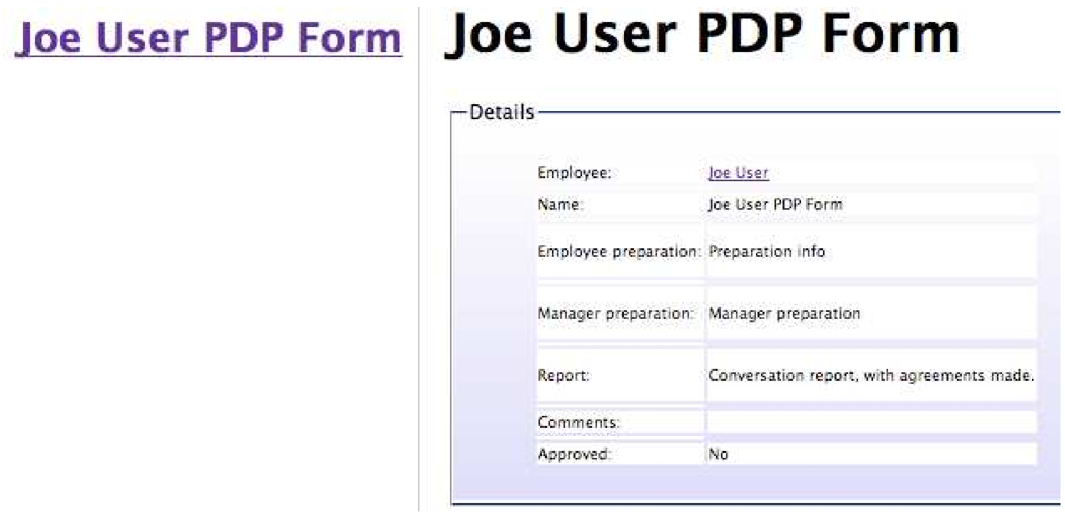

Joe User PDP Form

- Manager fill in form

Details

Manager preparation:	Manager preparation

After both the preparation forms have been filled in, the *write report* task becomes available for the manager. Filling in this report leads to the following view.


Joe User PDP Form

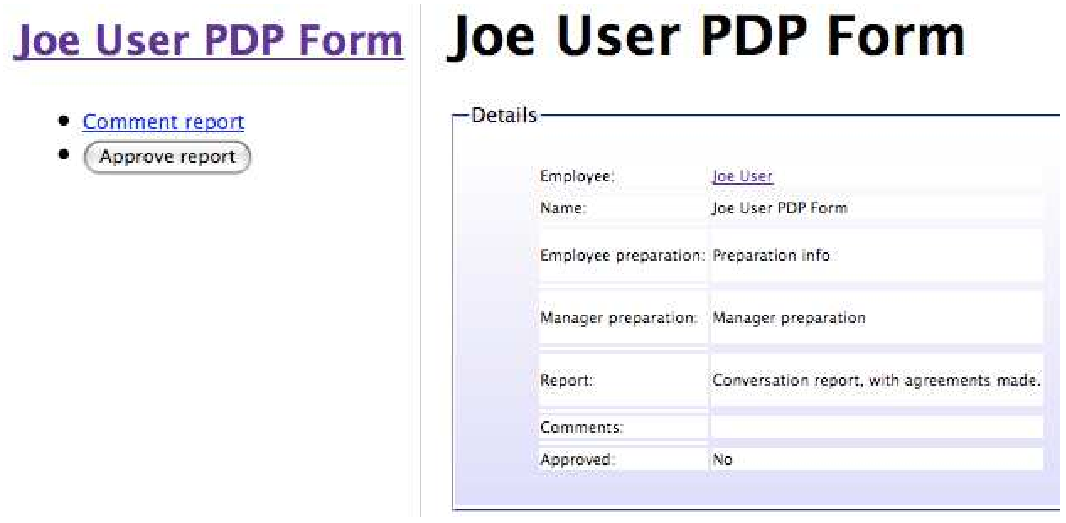
Joe User PDP Form

Details

Employee:	Joe User
Name:	Joe User PDP Form
Employee preparation:	Preparation info
Manager preparation:	Manager preparation
Report:	Conversation report, with agreements made.
Comments:	
Approved:	No

The sidebar on the left is empty, indicating that no more procedures need to be done by the manager.

Figure 4.3: Workflow Demonstration (Continued)



Joe User PDP Form

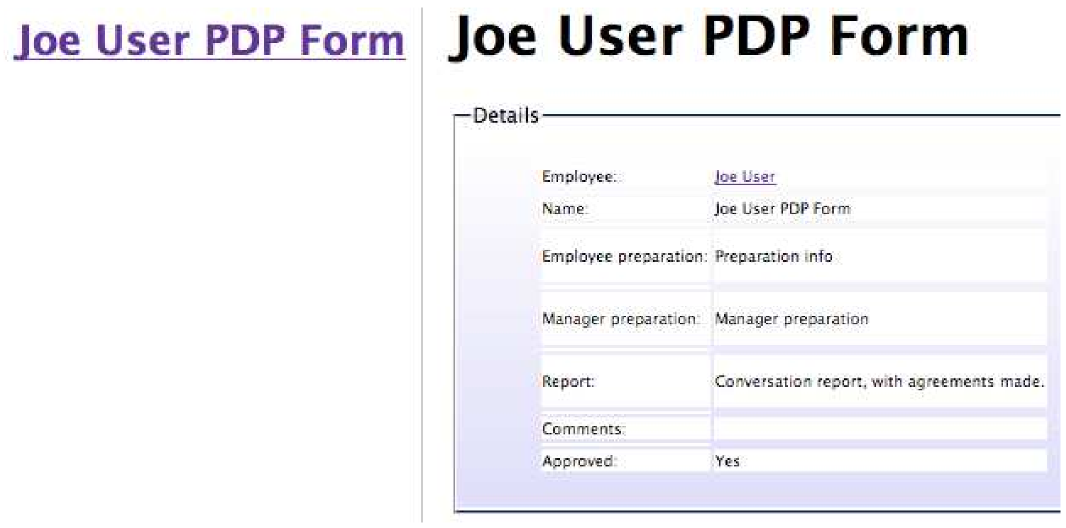
- Comment report
- Approve report

Joe User PDP Form

Details

Employee:	Joe User
Name:	Joe User PDP Form
Employee preparation:	Preparation info
Manager preparation:	Manager preparation
Report:	Conversation report, with agreements made.
Comments:	
Approved:	No

The employee however, has two new procedures that can be performed: *comment report* and *approve report*. Depending on the employee's choice, the workflow either finishes after approval of the report, or the *write report* task becomes available again in case the employee gives some comments on the report instead of approving it. The situation after choosing *approve report* is displayed in the next screenshot.



Joe User PDP Form

Joe User PDP Form

Details

Employee:	Joe User
Name:	Joe User PDP Form
Employee preparation:	Preparation info
Manager preparation:	Manager preparation
Report:	Conversation report, with agreements made.
Comments:	
Approved:	Yes

Figure 4.4: Workflow Demonstration (Continued)

4.2 Procedure Level: Flexibility

While the process description provides sufficient flexibility on itself to express most control-flow patterns, some patterns cannot be described without resorting to the lower level of procedures. Instead of using a process description for connecting procedures so that they are started at the right time, the procedures are rewritten so that their actions encode the desired behaviour.

An example of a pattern for which we need to specify custom procedure-level behaviour is derived from the example of a workflow for organizing an academic conference (see the example in section 2.5.2). We are aware of the fact that the automatic decision process for accepting academic papers used in this workflow would never be used in real-life, but it serves as a good example here.

Say the paper review process that is part of organizing a conference is as follows: a paper is reviewed by five different researchers, rating them *excellent*, *good*, *fair* or *poor*. After reviewing, the paper is accepted if it has either no poor ratings, or at least three excellent ratings. This means that after three of the five reviewers have rated the paper, the paper might already be accepted. If a paper is indeed rewarded with three excellent ratings, the workflow must send a response to the autors of the paper and continue with the rest of the workflow. Meanwhile, the two reviews that have not yet been received, can still be performed.

This pattern is not easily expressed using a process description. The closest description would be a construction in which five review procedures are performed in parallel, after which the decision is made to accept or reject a paper and continue with the rest of the workflow. This is caused by the fact that the process description does not have separate splits and joins, but rather different ways to at once specify a split and join (using `xor`, `and` or `+`).

However, the desired behaviour can be implemented using the following procedures.

```
extend entity Paper {
  reviews :: Set<Review> (inverse=Review.paper)
  excellentRatings :: Int
  accepted :: Bool }

function addReview(p : Paper) {
  var r : Review := Review{};
  p.reviews.add(r);
  p.persist();
  r.startPerformReview(); }

procedure paperReviewSplit(p : Paper) {
  do {
    addReview(p);
    addReview(p);
    addReview(p);
    addReview(p);
    addReview(p); } }

procedure performReview(r : Review) {
  view {
    main()
    define body() {
      form {
        table {
          row { input(r.rating) }
          row { action("Add review", do()) } } } } }
  do {
    r.persist();
    if (r.rating == "Excellent") {
      r.paper.excellentRatings = r.paper.excellentRatings + 1; }
    r.paper.persist();
    paperReviewJoin(r.paper); } }

function paperReviewJoin(p : Paper) {
  if (p.excellentRatings >= 3 && !p.accepted) {
    p.startNextProcedure();
    p.accepted = true;
    p.persist(); }
  var allFinishedReviews : Set<Review> :=
    [r
      for(r : Review in p.reviews
        where r.rating != "")];
  if (allFinishedReviews.count == 5) {
    var poorReviews : Set<Review> :=
      [r
        for(r : Review in p.reviews
          where r.rating == "Poor")];
    if (poorReviews.count == 0) {
      p.startNextProcedure();
      p.accepted = true;
      p.persist(); } } }
```

This example, in effect, creates multiple branches that operate independently. Each branch, when finished, will call `paperReviewJoin(p : Paper)`. This function will check the conditions for continuing with the rest of the workflow, and if fulfilling these conditions, starts the next procedure. It uses an attribute `p.accepted` to make sure that the next procedure is started only once.

This construction is illustrated by figure 4.5.

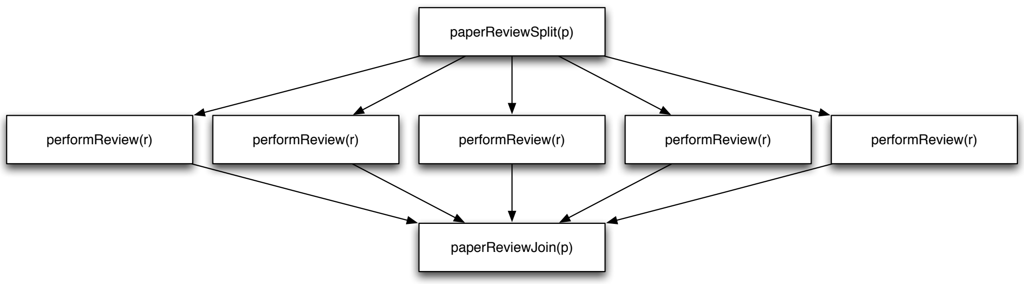


Figure 4.5: Procedure-level workflow specification

Chapter 5

Procedures in WebDSL

The implementation of WebWorkFlow consists of two distinct abstraction layers. The first layer is the *procedure layer*. In the procedure layer, abstractions are added for defining *procedures* for *entities*.

As shown in figure 5.1, the Procedural WebWorkFlow layer is built on top of several underlying abstraction layers, to which it transforms first. This means it can assume the presence of access control abstractions, templates, and everything contained in Core WebDSL.

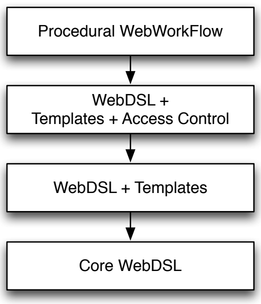


Figure 5.1: Procedures on top of WebDSL

A procedure is either an atomic amount of work - ranging from a simple acknowledgement or indication that some activity in the real world has been performed, to a complex algorithm - or an encapsulation of other procedures using a process description. In the latter case, it uses a higher abstraction layer to transform process descriptions into connections between procedures as shown in chapter 6. In this chapter, we restrict ourselves to looking at the procedure-level extensions to WebDSL.

If a procedure `exampleProcedure` is defined for an entity `exampleEntity`, this procedure can be *started* by calling `exampleEntity.startExampleProcedure()` and subsequently *performed*. A procedure may contain a *view* page definition, or else operate without view page, but instead be triggered by clicking a button. It is possible to specify that a procedure can only be done by a certain *actor*. Besides the actor, other *conditions* can be given that must hold for the procedure to be available. A procedure may contain an *action* that is executed when the procedure is performed. Procedures have additional sub elements that work like *event handlers*, executed when a procedure is enabled or disabled or after performing a procedure.

In addition, instead of being an atomic amount of work, procedures can contain a process description, in which case they activate several other procedures connected in a way described in this process description, and wait for them to be performed. As mentioned, this will be explained in chapter 6, as it belongs to a higher abstraction layer.

If any procedure is defined for an entity `exampleEntity`, a task page `exampleEntityTasks()` is generated, as well as a template `exampleEntityTaskList()` that is used on this task page. This page displays links to any objects of type `exampleEntity`, for which the currently logged in user can perform any *task* or procedure, and also provides links to these procedure pages. In case any procedure is defined at all, a task page `allTasks` is generated, that displays the same for all entities on which the current user can perform a task.

In the remainder of this chapter, we will first explain the different sub elements of procedures in more detail in section 5.1. After this, we will cover the procedure execution model in section 5.2, in which we will also explain the different event handlers that are available. Next, we will discuss how this abstraction layer is implemented by looking at the transformation to WebDSL. After this, we show two different ways to connect procedures to form a workflow. Finally, we will look at a method to enable procedures to have multiple instances.

5.1 Procedural Language Extension

In this section, the different sub elements of procedures are explained in more detail. We cover *actors*, *additional constraints*, *view definitions* and *actions*, after which we explain what *automated procedures* are. When possible, examples from the PDP Meeting example in chapter 4 will be used to illustrate the use of the sub elements.

The additional sub elements of procedures that function as *event handlers*, will be explained in the next section.

5.1.1 Actors - who

An *actor* sub element consists of a boolean expression to verify the current user's identity. If a procedure contains an actor sub element, the procedure is only displayed if the currently logged in user fulfills the condition it contains. For instance, the following procedure (from the PDP Meeting example in chapter 4) is only visible, and can only be performed, by the manager of the user referenced by the `PdpMeeting` object for which the procedure is enabled:

```
procedure writeReport(p : PdpMeeting) {
  who { securityContext.principal == p.employee.manager }
  ...
}
```

The `who {}` construct normally draws on the access control sub language in WebDSL.

5.1.2 Additional Constraints - when

It is possible to define additional constraints that cause a procedure to be available only under certain circumstances. For instance, the next example is only displayed in case the *employeePreparation* property of *p* is still empty.

```
procedure employeeFillInForm(p : PdpMeeting) {
  when { p.employeePreparation == "" }
  ...
}
```

5.1.3 Action - do

For specifying an action to be executed when a procedure is performed, the `do {}` sub element can be used.

An example is the following procedure:

```
procedure performReview(r : Review) {
  view {
    form {
      input(r.rating)
      action("Add review", do())
    }
  }
  do {
    r.persist();
    if (r.rating == "Excellent") {
      r.paper.excellentRatings = r.paper.excellentRatings + 1;
    }
    r.paper.persist();
  }
}
```

This example gives a user the opportunity to submit a review. The *view* sub element contains a form with an input field for the review rating. The *do* sub element subsequently persists the review and optionally increments the *excellentRatings* counter. As can be seen in this example, the *do* sub element can be referenced within *view* as an action called `do()`.

If a procedure does not have an associated action, nothing will happen. Procedures without actions are often used when the procedure is a part of a workflow containing multiple procedures. To perform a procedure without an action, may signify the occurrence of an event in the real world, or may just be used to halt the workflow until a certain moment.

5.1.4 View definition - view

The view definition enables the developer to make a custom design for any procedure page. Usually, it contains a form with some input fields for the purpose of user input.

Besides giving a custom page, it is also possible to use `derive procedurePage`, as shown in this example:

```
procedure employeeFillInForm(p : PdpMeeting) {
  who { securityContext.principal == p.employee }
  view {
    derive procedurePage from p for (employeePreparation) {
      title{"Fill in employee form"}
```

```
        header{"Fill in employee form"}
      }
    }
  }
```

When using `derive procedurePage`, a sidebar is automatically displayed, containing links to all procedures that are available for the current object. This example will print a page with a form to edit `p`'s property `employeePreparation`. When using `derive procedurePage`, it is not necessary to add a `do {}` sub element, because it is generated automatically.

If no view definition is given, the procedure will not have a corresponding page to view when performing the procedure. Instead, when displaying the task list, a button will be displayed instead of a link to the procedure page. This button will immediately trigger the action.

5.1.5 Automated Procedures

A special type of procedure is the *automated procedure*. An automated procedure is a procedure without a *view* definition, that is performed immediately after it is started, without needing any human intervention. Also, it cannot have any *who* or *when* conditions for it to be performed. These automated procedures are very convenient for tasks that do not need any user input or intervention.

Usually, when defining a procedure that contains a process description, it is defined as an automated procedure. If this is not done, the user first has to click a button to 'perform' this procedure, after which the procedures referenced in its process description will become available. Normally, we will want the procedures referenced in the process description to be available immediately.

An example of an automated procedure in which a confirmation is automatically sent by email:

```
  auto procedure sendEmailConfirmation(s : Subscription) {
    do {
      ...
    }
  }
```

5.2 Procedure Execution Model

Besides *who*, *when*, *do* and *view*, there are several *event handlers*, that can be used to customize procedures even further. The available events are *enabled*, *disabled*, *done* and *processed*. The execution of the events is illustrated in figure 5.2. This figure shows two separate flows of control.

The right side of the figure shows a short flow of execution that describes what happens when the procedure is disabled.

The normal flow of execution is displayed on the left side of the figure. This shows the flow of execution from enabling the procedure up to the point where all tasks have been performed, after *processed*. We will describe this execution model in more detail here.

When a procedure is started using `object.startExampleProcedure()`, two things happen. First, using the entity's function `object.prepareExampleProcedure()`, the necessary procedure status objects are constructed. Second, by a call to `object.exampleProcedure.enable()`, the procedure is activated. This causes the procedure to be available for being performed by its actors, if the *when* clause evaluates to true. After being enabled, the event handler *enabled* is called.

If the procedure has a *view* page, viewing this page is the next thing that happens. This step is initiated by the user. If the user chooses to perform the procedure, the associated *do* action is executed. After this, the event called *done* is fired.

After executing the procedure's action, an optional *process description* is executed. This means that the procedures that are referenced in the process description are started, and the procedure waits until the entire process description is executed, before firing the last event called *processed*.

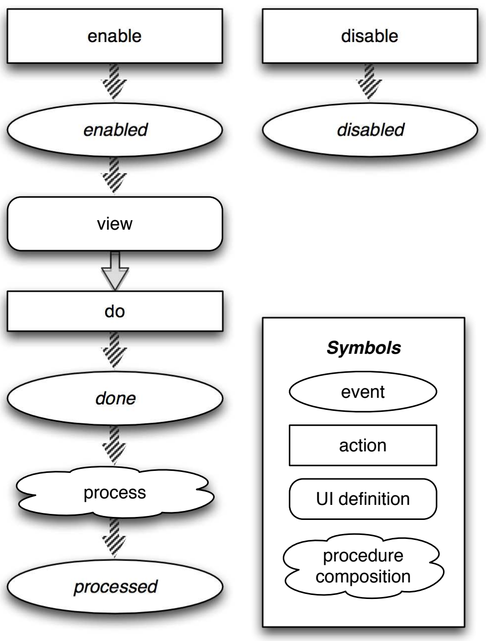


Figure 5.2: WebWorkFlow Execution Model

There is an important difference between what happens when an event handler is given when defining a procedure, and what happens when defining other elements of a procedure. When defining event handlers, the default event handlers are extended with the given code. When defining other elements, the default elements are overridden. For instance, if something should happen after enabling a procedure, it must be specified using `enabled()`, and not using `enable()`. This is because the default behaviour of `enable()` takes care of setting the procedure status entity's `isEnabled` variable, as well as firing the `enabled()` event. This gives the possibility to overwrite a default element like `enable()` and `disable()` to implement custom behaviour. The differences between extending en overriding can be better understood by looking at what is generated when transforming procedures to WebDSL code in 5.3

5.3 Transformation to WebDSL

Because the Procedural WebWorkFlow layer uses all underlying WebDSL abstraction layers for its execution, the only thing we have to add to the WebDSL compiler is a *transformation from procedures to regular WebDSL* (including access control).

In this section, we explain the different elements that are generated for procedure definitions. In the course of explaining the different elements, we use an example procedure definition:

```
procedure proc(object : Example) {
  who  { securityContext.principal == object.owner }
  when { !object.finished }
  view { form {
```

```
            input(object.title)
            action("Save title", do())
          } }
    do   { object.finished = true;
          object.persist(); }
  }
```

The remainder of the section is organized as follows. First, we describe the entity model extension that is made for translating procedure definitions. After that, we discuss the extensions that are made to the entity for which a procedure is defined, and finally we will cover the page generation for procedures.

5.3.1 Generating Procedure Status Entities

For saving procedure-bound data, we create an entity for every defined procedure. For `proc`, this will be the entity *ProcProcedureStatus*. This status entity will inherit from the main *ProcedureStatus* entity, displayed here:

```
  entity ProcedureStatus {
    name :: String := "Procedure status"
    isEnabled :: Bool
    isProcessed :: Bool
    function enable() {
      this.isEnabled := true;
      this.persist();
      this.enabled();
    }
    function enabled() {}
    function done() {}
    function processed() {
      this.isEnabled := false;
      this.isProcessed := true;
      this.persist();
    }
    function disable() {
      this.isEnabled := false;
      this.persist();
      this.disabled();
    }
    function disabled() {}
  }
```

There is a function here for every activity or event that is explained in the Procedure Execution Model, except for *view* and *do*. *view* is not a function, but a page definition, and as such will show up later, when generation of procedure pages is discussed in section 5.3.4. Because the action defined in `do {}` is not added to the entity *ProcProcedureStatus* but to the procedure page, it does not show up in *ProcedureStatus*.

The `enable()` function triggers the execution of `enabled()`. Like this, the `disable()` function triggers the execution of `disabled()`. However, `done()` is not triggered necessarily by `do {}`. As explained in section 5.1, not all procedures have an associated action. Therefore, the execution of `done()` is triggered in one of two ways. If a `do {}` element is defined, a call to `done()` is added to it. If no action is defined, `done()` is called from `enable()`.

Auto procedures have yet another way of performing `do {}`. Auto procedures, as stated, cannot have a view definition. Instead, the action in auto procedures is performed within *enable*, so that it is executed immediately when the action becomes available.

The `isEnabled` property of *ProcedureStatus* indicates whether the procedure can be available or not. It is set to true during `enable()` and set to false after the procedure has been performed completely (which is at `processed()`), or when the procedure is wilfully disabled. When a procedure has been performed completely, the value of `isProcessed` will be set to `true`.

If a `procedure proc(o : ExampleEntity)` was declared for an example entity `ExampleEntity`, the following procedure status entity would result.

```
entity ProcProcedureStatus : ProcedureStatus {
  o -> ExampleEntity()
  ...
}
```

The object for which the procedure is run, can therefore be accessed through the procedure status object, by using the variable name used at the location of the procedure declaration.

5.3.2 Extending the Entity

Procedures are generated for existing entities. To abstain from cluttering up the domain data model, we keep the procedure-bound data in a separate entity, deriving from `ProcedureStatus`. However, to link a procedure to an entity, we have to add a reference to the procedure status entity:

```
extend entity Example {
  proc -> ProcProcedureStatus
}
```

5.3.3 Starting a Procedure

For the purpose of *preparing* and *enabling*, or *starting* a procedure, we extend `Example` with the following functions:

```
extend entity Example {
  function prepareProc() {
    if (this.proc == null) {
      this.proc := ProcProcedureStatus{};
      this.proc.o := this;
      this.proc.persist();
      this.persist();
    }
  }
  function startProc() {
    this.prepareProc();
    this.proc.enable();
  }
}
```

`prepareProc()` will prepare the entity for enabling the procedure called proc, by creating a `ProcProcedureStatus` object, that is subsequently used for all procedure-related data. The function can safely be called twice, as it checks if the procedure status entity has already been constructed.

Instead of first preparing a procedure and subsequently enabling the procedure, it is possible to use the function `startProc()`. This function will be used for starting a procedure in most situations.

5.3.4 Generating Pages

For every procedure, a procedure page is generated. For with a *view* definition, this page will be viewed by the user just before performing the procedure. Procedures without *view* are performed instantly when their procedure page is requested, provided the conditions for its execution are met.

When generating a procedure page, three things have to be considered. Firstly, the procedure's *view* definition, secondly, the *action*, and thirdly, the conditions for the procedure to be performed.

If a procedure has no *view* definition, it is generated as follows:

```
define page proc(object : Example) {
  init {
    <optional action>
    object.proc.done();
    goto example(object);
  }
}
```

In this way, when requesting the page, it will first perform the optionally given action. It will subsequently trigger `done()`. Afterwards, it will redirect to the object's view page.

If only this page was generated, anyone would be able to perform any procedure at any time. To make sure that the page is only available when it should be, we have to take the following into account:

- A procedure page can only be requested when the procedure is enabled;
- If an actor is specified, the page can only be requested by a logged in user that fulfills the *who* condition;
- If additional conditions are specified, the page can only be requested if the *when* condition is fulfilled.

To enforce these constraints, we use WebDSL's action control language. To construct the rule for showing the page, we only have to make sure the above conditions evaluate to `true`. This is done by adding an access control rule as is displayed for our example in the following code snippet:

```
access control rules

  rule page proc(object : Example) {
    object.proc.isEnabled && (securityContext.principal == object.owner) && (!object.finished)
  }
```

5.4 Connecting Procedures

Procedures are activities that can be performed for an object. They introduce a process-oriented element into WebDSL. However, to use procedures to form a workflow, we must connect different procedures. This can be done in several ways, two of which we will explain in this section. The first method is using *shared data*. The second method is by *letting procedures start other procedures*.

5.4.1 Connecting Procedures Using Shared Data

Using constraints (`when {}`), it is possible to create an extensive process in which multiple procedures are executed in a specified order. This can be done by extending the entity for which the procedure is defined with a status attribute, and by changing this status attribute during the course of executing several procedures, as shown in the next example.

```
extend entity Example {
  status :: String
}

auto procedure workflow(object : Example) {
  do {
    object.status := "step1";
    object.persist();
    object.startStep1();
    object.startStep2();
  }
}

procedure step1(object : Example) {
  when { object.status == "step1" }
  done {
    object.example := "step2";
  }
}

procedure step2(object : Example) {
  when { object.status == "step2" }
}
```

By extending the entity `Example` with a status property, we introduce data that is shared between different procedures. This example implements a workflow of two simple, sequential steps. These steps are made available by starting the procedure called *workflow*. This workflow starts both procedures after setting the status of the object to `"step1"`. This results in both procedures being started, but only one begin available, as only step1's condition holds. After step1 is performed, it alters the status attribute so that step2 becomes available.

5.4.2 Letting Procedures Start Other Procedures

Instead of using shared data to encode status information, it is also possible to connect procedures by letting them enable each other. An example of this is given in the next code snippet.

```
procedure step1(object : Example) {
  ...
  done {
    object.startStep2();
  }
}

procedure step2(object : Example) {
  ...
}
```

This method is much simpler, but still very powerful. The only advantage of using a status property is that there is an explicit registration of the present state, which can help when debugging. But the latter method is much more concise.

As will be shown in chapter 6, this method is the one that is used when translating process descriptions to connect procedures.

5.5 Multiple Instances

Entities for which procedures are defined, are extended with a reference to only one procedure status entity. This means that at any time, only one instance of a procedure can be started for every object of this entity type. However, there is a way around this, which we will show here.

Instead of defining a procedure for entity *Example*, we could define a procedure for an entity *ExampleInstance*. If we manually extend *Example* with a reference to a *set* of *ExampleInstances*, we can start multiple procedures for the same entity, which is demonstrated in the following example.

```
entity ExampleInstance {
  example -> Example
}

procedure proc(object ExampleInstance) {
  ...
}

extend entity Example {
  instances -> Set<ExampleInstance> (inverse=ExampleInstance.example)
  function startNewProc() {
    var instance : ExampleInstance := ExampleInstance{};
    this.instances.add(instance);
    this.persist();
    instance.startProc();
  }
}
```

This example implements the solution we discussed. It also provides `startNewProc()`, an entity function that creates a new instance for which procedure *proc* is started, and adds it to the set of instances.

Chapter 6

Transforming Process Descriptions

After introducing the Procedural WebWorkFlow abstraction layer in chapter 5, we now continue by showing the highest layer of abstraction: Process WebWorkFlow. As before, constructs in this abstraction layer are merely translated to constructions in the abstraction layer beneath, in this case the procedural layer.

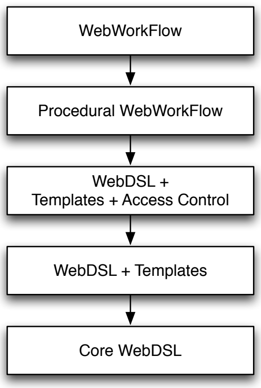


Figure 6.1: The full abstraction stack including WebWorkFlow

Process descriptions form what can be called the heart of WebWorkFlow. They provide constructs for a concise expression of control-flow patterns to enable a straightforward definition of workflow processes. While they lift the burden for procedures to enable each other, the procedures still have to be defined separately, when they are referenced in process descriptions. The process level only deals with control-flow, while the procedural level still takes care of *who* and *when* procedures are available, what the *view* of a procedure page looks like, and what *actions* are executed when a procedure is performed.

WebWorkFlow process descriptions are meant to cover the majority of the regularly used control-flow patterns. If at any time, however, the constructs provided by process expressions do not provide enough flexibility for a certain workflow, the developer can resort to manually specify connections between procedures, and bypass the process level.

The process language is different from most workflow languages in two aspects.

1. WebWorkFlow process definitions are textual, rather than graphical.

2. WebWorkFlow does not use separate splits and joins. Instead, it provides different ways for branching the process without needing separate splits and joins. This results in a very clear view on the process.

The translation of process expressions to procedure level abstractions is made by a transformation in which the separately defined procedures are altered to conform to the process expression's control-flow specification. In this chapter, we will cover all different control-flow constructs provided by WebWorkFlow, as well as their translation to procedure-level constructs. First, we will elaborate on the way procedure calls are handled in WebWorkFlow. Next, we will introduce our approach to transforming process descriptions to procedure-level code.

We will use the remainder of this chapter for describing the various constructs, as well as their transformation to procedure-level constructs. We will start with the simplest of constructions: sequential composition. Next, we will cover conditional execution by treating the `if {} else {}` construct. After this, we will tackle parallel execution of procedures, describing the operators `and` and `or`. Then we will show two different ways of expressing loops in processes, introducing `while {}` and `repeat {} until`. Finally, we will introduce a special construct called *deferred choice*, that is used for creating two branches in the process, of which one is chosen, but only when any task within that branch is performed.

6.1 Handling Procedure Calls

To implement sequential execution of procedures as well as branching, we need to be able to call procedures as if they were functions. However, some process descriptions inhibit parallellism, which makes it impossible to have a view on execution of procedures that is synchronized. Instead, we have to allow for asynchronous execution of procedures. We solve this problem by splitting the call of a procedure in two pieces: starting a procedure, and resuming execution of the process after the particular procedure has finished execution. The latter half of this action is performed by providing a callback function that is called by the procedure that is executed, after it has finished. For the process engine to be able to continue execution after a procedure has been performed, it needs a way of representing the state of execution to remember what to do when resuming execution.

To make this possible, we have to make several additions to the *ProcedureStatus* entity displayed in chapter 5. These additions are shown in the following code snippet, and explained afterwards.

```
extend entity ProcedureStatus {
  caller -> ProcedureStatus
  returnstate :: Int
  branch :: Int
  function enable (c:ProcedureStatus, r:Int, b:Int) {
    // replaces previous version
```

```
      this.isEnabled := true;
      this.caller := c;
      this.branch := b;
      this.returnstate := r;
      this.persist();
      this.enabled();
    }
    function next(state:Int) {}
    function processed() {
      // previous version extended with ...
      if (this.caller != null) {
        if (this.branch != null) {
          this.caller.notifyOfActivity(this.branch);
        }
        this.caller.next(this.returnstate);
      }
    }
  }
```

The *next* function is used to implement sequential execution of procedures. It takes an integer argument used to pass the state of execution. When procedures are called within a process description, instead of the regular `enable()` without arguments, enable is called with three arguments: a caller, a return state and a branch. The branch is explained later. The caller is used to save a reference to the procedure status entity from which the call to this procedure originated. The return state is used for saving the state of the caller's execution, so when resuming execution, it knows what to do next.

A new *processed* function is also given. If the procedure was called by another procedure, it makes sure that the callback function of that procedure is called with its returnstate as a parameter.

6.2 Transformation Approach

We are concerned with the transformation between process-level WebWorkFlow and procedure-level WebWorkFlow, as can be seen in figure 6.2

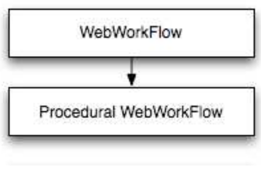


Figure 6.2: Process Transformation

Our general approach to transforming process expressions to procedures is using a single traversal of the process expression tree, in which the following operations are performed:

1. All complex constructs used in (a part of) the process description are transformed to single procedures. Within these procedures, the behaviour of the original construct is implemented

using code at the procedure level. Complex constructs are all constructs except sequential composition.

2. The simplified constructs are sequentially connected.

Complex constructs may contain only a single procedure (such as `if (`*`condition`*`) { a(o) }`), in which case no additional transformation of sub-expressions needs to be done. If not, the sub-expression it contains is handled first, using the same algorithm.

6.3 Sequential Composition

Sequentially connecting procedures in process expressions is done only after the simplification of more complex process constructs has taken place. This means that, when sequentially connecting the process expressions, we can assume that all we will encounter is simplified procedures.

For example, look at the following process expression:

```
process {
  step1a(o) and step1b(o)
  ; if (o.count > 2) {
      step2(o)
    }
}
```

This process expression will be simplified so that when making sequential connections between all simplified procedures in this process expression, we will find the process expression changed to include the simplified procedures rather than the original ones:

```
process {
  andProc0(o)
  ; ifProc0(o)
}
```

Connecting procedures is done in the following way. For each procedure, an entry is made in the next procedure of the encapsulating procedure, in which the procedure is prepared and, subsequently enabled. When enabling procedures, a return state is provided, which allows the procedure to enable the next procedure in the process using its caller's next function. For the example shown, the next function will be:

```
function next(state:Int) {
  if (state == 0) {
    this.o.prepareAndProc0();
    this.o.andProc0.enable(this as ProcedureStatus, 1, 1);
  }
  if (state == 1) {
    this.o.prepareIfProc0();
    this.o.ifProc0.enable(this as ProcedureStatus, 2, 1);
  }
  if (state == 2) {
    this.processed();
  }
}
```

For now, we ignore the last argument to *enable*. When enabling *andProc0*, the second argument - the return state - is 1, so that when the flow of control is given back to *next*, the next procedure will be enabled, in this case *ifProc0*. After all procedures in a list of sequentially ordered procedures have been performed, the encapsulating procedure's *processed* function is called, and the process will finish.

6.4 Conditional Execution: If

Before any process expression besides sequential connections can be used in the final sequential connecting, they have to be simplified. We will explain these simplifications of process expressions in the remainder of this chapter, starting with *conditional execution*.

When simplifying process expressions, a new procedure is generated to replace the expression. This new procedure takes the responsibility to guide the executions of all procedures referenced within the original process expression.

Conditional execution of procedures is done by evaluating an *expression*, after which the *if* branch is activated in case the expression evaluates to `true`, or else either the optional *else* branch is activated or the simplified *if* procedure ends with a call to *processed* in case no else branch is given.

We will show the simplification of conditional execution using the following example:

```
process {
  if (o.expression) {
    a(o)
  } else {
    b(o); c(o)
  }
}
```

This is simplified to form the next expression:

```
process {
  ifProc0(o)
}
```

What really matters is what happens within the generated *ifProc0*. Its status entity is displayed in part here:

```
procedure ifProc0ProcedureStatus : ProcedureStatus {
  ..
  function enabled() {
    if (this.o.expression) { this.next(0); }
    else                   { this.next(2); }
  }
  function next(state:Int){
    if (state == 0) {
      this.o.prepareA();
      this.o.a.enable(this as ProcedureStatus, 1, 1);
    }
    if (state == 1) { this.processed(); }
    if (state == 2) {
      this.o.prepareB();
      this.o.b.enable(this as ProcedureStatus, 3, 2);
    }
    if (state == 3) {
      this.o.prepareC();
      this.o.c.enable(this as ProcedureStatus, 4, 2);
    }
    if (state == 4) { this.processed(); }
  }
  function disabled() {
    if (this.o.a != null) { this.o.a.disable(); }
    if (this.o.b != null) { this.o.b.disable(); }
    if (this.o.c != null) { this.o.c.disable(); }
  }
}
```

When the procedure is enabled, it evaluates the condition. Based on the outcome, it calls either `next(0)` or `next(2)`. When looking at *next*, we can see that it contains two separate, sequentially connected, flows of control. The first, belonging to the *if* branch, starts with state `0` and ends with state `1`, in which *processed* is called. The second, belonging to the else branch, starts with state `2` and ends with state `4`.

The disabled function is also extended. When disabled, the *ifProc0* procedure makes sure that all potentially started procedures are disabled, as well.

6.5 Parallel Execution: And and Xor

The expressions `and` and `xor` are used to model parallel execution of procedures. For both expressions, two branches are enabled at the same time. The behaviour when branches finishes, however, is what distinguish `and` from `xor`. When using `and`, a finishing branch will not cause the workflow to continue immediately. Instead, it waits for the other branch to finish, as well, after which it returns control to its caller. `xor` behaves somewhat differently. When any of the two branches finish, the other branch is disabled, and the workflow continues.

Both constructs are binary operators. They can be used in a bigger expression, however. The priority of `and` is higher than `xor`'s priority, and as such, `and` and `xor` can be compared to multiplication and addition, respectively, in ordinary mathematics.

We will now show the generated code for the two process expressions when simplifying them.

6.5.1 And

The following example demonstrates the simplification of `and`.

```
  procedure andProc0ProcedureStatus : ProcedureStatus {
    ..
    otherBranch :: Int
    function enabled() {
      this.next(0);
      this.next(2);
    }
    function next(state:Int){
      if (state == 0) {
        this.o.prepareA();
        this.o.a.enable(this as ProcedureStatus, 1, 1);
      }
      if (state == 1) {
        if (this.otherBranch == 0) {
          this.otherBranch := 1;
          this.persist();
        } else {
          this.otherBranch := 0;
          this.processed();
        }
      }
      if (state == 2) {
        this.o.prepareB();
        this.o.b.enable(this as ProcedureStatus, 3, 2);
      }
      if (state == 3) {
        if (this.otherBranch == 0) {
          this.otherBranch := 1;
          this.persist();
        } else {
          this.otherBranch := 0;
          this.processed();
        }
      }
    }
  }
```

Instead of choosing between two branches, both branches are started in `enabled()`.

When one of the two branches finishes, it checks if the other branch has already finished. If not, it sets `this.otherBranch` to `1`. If it has, it changes `this.otherBranch` back to `0` and ends the procedure by calling *processed*.

The omitted function `disabled()` is identical to the function shown in condition execution.

6.5.2 Xor

There are few differences between the simplification of `and` and `xor`. The only difference is the way it reacts to branches finishing. Consider the following parts of the *next* function, if `xor` is used in the previous example, instead of `and`:

```
  function next(state:Int) {
    ...
    if (state == 1) {
      if (this.p.b != null) {
        this.p.b.disable();
      }
      this.processed();
    }
    ...
    if (state == 3) {
      if (this.p.a != null) {
        this.p.a.disable();
      }
      this.processed();
    }
    ...
  }
```

Instead of waiting for the other branch to finish, if one branch finishes, the contents of the other branch is disabled and the procedure ends.

6.6 Loops: While and Repeat Until

WebWorkFlow provides two different constructs for loops in processes: `while(condition) {}` and `repeat {} until`. The `while` loop takes a condition on which to start another iteration, functioning in the same way as while loops in most programming languages. Repeat, however, works slightly different: the loop is repeated until, at the end of an iteration of the loop, the *until* procedure is performed.

6.6.1 While

The `while` loop is demonstrated in the following example:

```
  process {
    while(o.condition) {
      a(o); b(o)
    }
  }
```

Simplifying this process description causes the generation of a procedure *whileProc0*. Its procedure status object is partially displayed here, after which we will explain its characteristics.

```
  entity WhileProc0ProcedureStatus : ProcedureStatus {
    function enabled() {
      if (o.condition) {
        this.next(0);
      } else {
        this.processed();
      }
    }
    function next(state:Int) {
```

```
    if (state == 0) {
      this.o.prepareA();
      this.o.a.enable(this as ProcedureStatus, 1, 1);
    }
    if (state == 1) {
      this.o.prepareB();
      this.o.b.enable(this as ProcedureStatus, 2, 1);
    }
    if (state == 2) {
      this.enable();
    }
  }
}
```

When enabling the procedure, the given condition is evaluated. If it evaluates to `true`, the loop is started. After finishing the sequentially ordered procedures referenced in the loop, the procedure enables itself, which causes it to start at the beginning. This process only stops if, when the condition is evaluated at the start of each iteration, it evaluates to `false`. In that case, *processed* is called and the procedure ends.

6.6.2 Repeat Until

An example of the use of *repeat until* is shown here.

```
process {
  repeat {
    a(o); b(o)
  } until c(o)
}
```

The behaviour of this expression is as follows. `a(o); b(o)` is repeated until `c(o)` is performed. This can only be done at the end of an iteration of the loop. If, however, the user decides not to perform `c`, but to execute the loop again by performing `a`, the *until* clause is disabled again.

To implement this, we need a mechanism to inform us of any procedure being performed within the loop. For this cause, we add `notifyOfActivity(branch : Int)` and `cascadeNotification()`. As could be seen in code excerpts earlier, when *processed* is executed, if *caller* is non-empty, a call to *notifyOfActivity* on the caller is performed as well: `this.caller.notifyOfActivity(this.branch);`. This function might use the notification to do something, but in any case passes the message on by calling `cascadeNotification()`, which informs its caller of the activity. In this way, the message travels all the way upward. The branch argument is used to distinguish between branches in determining in which branch the activity took place, so activity in the until branch will not be mistaken for activity in the repeat branch. This will also be useful later, in simplifying the *deferred choice* construct in section 6.7.

Simplifying the example process expression generates the following procedure status entity:

```
entity RepeatUntilProc0ProcedureStatus : ProcedureStatus {
  function notifyOfActivity (branch : Int) {
    this.cascadeNotification();
    if (branch == 1) {
      if (this.o.c != null) { this.o.c.disable(); }
    }
  }
  function cascadeNotification() {
    if (this.caller != null && this.branch != null) {
      this.caller.notifyOfActivity(this.branch);
    }
  }
  function enabled() {
    this.enableRepeat();
  }
  function enableRepeat() {
    this.next(0);
  }
  function enableUntil() {
    this.next(3);
  }
  function disableRepeat() {
    if (this.o.a != null) { this.o.a.disable(); }
    if (this.o.b != null) { this.o.b.disable(); }
  }
  function next(state:Int) {
    if (state == 0) {
      this.o.prepareA();
      this.o.a.enable(this as ProcedureStatus, 1, 1);
    }
    if (state == 1) {
      this.o.prepareB();
      this.o.b.enable(this as ProcedureStatus, 2, 1);
    }
    if (state == 2) {
      this.enableRepeat();
      this.enableUntil();
    }
    if (state == 3) {
      this.o.prepareC();
      this.o.c.enable(this as ProcedureStatus, 4, 2);
    }
    if (state == 4) {
      this.disableRepeat();
      this.processed();
    }
  }
}
```

When starting `repeatUntilProc0`, the repeat loop is enabled first, by calling next with state `0`. After both procedures in the repeat loop have been performed, both the repeat loop and the until procedure are enabled. If the until procedure is performed rightaway, the procedure will disable its repeat branch, and finish by calling *processed*. If the user decides to perform the first procedure referenced within the repeat loop, the function *notifyOfActivity* will be called with branch `1` as an argument, which will cause the until procedure to be disabled.

6.7 Deferred Choice

Besides `and` and `xor`, there is a third operator that exhibits parallel execution. Using the construct called *deferred choice* will start two branches at the same time, of which only one will be actually executed. The choice between the branches is made at the moment a user performs a procedure in one of the two branches.

We will use the following example to demonstrate the transformation to procedures that is done when translating a *deferred choice* construct.

```
entity PlusProc0ProcedureStatus : ProcedureStatus {
  function enabled() {
    this.next(0);
    this.next(2);
  }
  function next(state:Int) {
    if (state == 0) {
      this.o.prepareA();
      this.o.a.enable(this as ProcedureStatus, 1, 1);
    }
    if (state == 1) {
      this.processed();
    }
    if (state == 2) {
      this.o.prepareB();
      this.o.b.enable(this as ProcedureStatus, 3, 2);
    }
    if (state == 3) {
      this.processed();
    }
  }
  function notifyOfActivity ( branch : Int ) : Void
  {
    this.cascadeNotification();
    if (branch == 1) {
      if (this.o.b != null) {
        this.o.b.disable();
      }
    }
    if (branch == 2) {
      if (this.o.a != null) {
        this.o.a.disable();
      }
    }
  }
}
```

At the start of this procedure, both `a` and `b` are started. The two sequential flows of control in *next* work without influencing each other. The function *notifyOfActivity* is used to enforce a choice between the two branches. The two branches are enabled with two different values for the *branch* argument in the call to *enable*. This will let the call to *notifyOfActivity* know which of the branches has been chosen. If branch `1` is chosen, branch `2` will be disabled, and vice versa.

Chapter 7

Case Study: LIXI Workflows

Both WebDSL and our WebWorkFlow extension to WebDSL had not been tested using real-life applications, yet. Therefore, both would benefit from a case study involving a real-life application, to present some proof of their usability. To test the applicability of WebWorkFlow in real-life circumstances, we have performed a case study in which an application that is actually used, is modeled using WebWorkFlow. The case study involves an existing *property valuation* workflow, designed in the context of the *LIXI* project: *The Lending Industry XML Initiative*.

In this chapter, we will present the LIXI workflow, as well as an implementation of the workflow using WebWorkFlow. We will start by explaining the LIXI project's background, after which we will talk about the actual valuation workflow, including its current implementation. Next, we will show our implementation of the valuation workflow, in the course of which we will also comment on choices we had to make during the process. In this section, we will also demonstrate the generated workflow application. After this, we will show some improvements we made to the WebWorkFlow language and compiler as a result of the case study. Finally, we will examine the results of our work.

7.1 Motivation and Background

The lending industry is a big industry involving thousands of different parties, including banks, brokers and borrowers. These parties all have their own information systems, to help them organize their work.

Often, transactions involve multiple parties, all having their own information system. These transactions, however, can usually only be monitored from the point of view of one of the parties, as the information systems have no information about events beyond their own boundaries. Communication between parties therefore happens mainly in the form of e-mail and telephone calls. Information is not centrally managed. If any communication error occurs, it may cost days to detect and resolve the problem.

To have an integrated view on the activities of multiple parties, the industry would benefit from a standard method for communicating.

The LIXI project seeks to standardize processes in the context of the *lending industry*. This includes banks, brokers, borrowers and property valuators. As LIXI's website states,

> The Lending Industry XML Initiative (LIXI) is a non-profit, independent industry organisation established to develop e-Commerce standards and remove barriers to electronic data exchange within the Australian lending industry. [34]

These XML communication standards are developed by research groups formed by members of LIXI. NICTA has a special status within LIXI and forms a research group on its own. They have been involved in developing process models and in developing a *reference implementation*. While LIXI, having a very heterogeneous group of members, does not want to enforce an implementation standard as reference architectures usually do, they do want to provide some guidelines to implementing a system that conforms to the LIXI standards. NICTA has developed APIs and code snippet libraries forming a reference implementation, sometimes called *reference architecture*.

NICTA has been involved with LIXI since 2004, and are now involved in doing process modeling for a National Electronic Conveyancing System (*NECS*), a system that provides a way of transferring properties between owners that is uniform throughout all states in Australia, while presently, all states only have their own system of conveyancing. This project will run until at least 2011.

7.2 Valuation Workflow

In this section, we describe the valuation workflow in detail. We start by explaining the roles involved. Then, we present an overview of the workflow process. Finally, we will shortly explain how the current implementation of the workflow works.

7.2.1 Roles

A property valuation process involves several parties, namely a *client*, an *administrative employee*, a *valuer*, and a *manager*.

The *client* is usually a bank, broker or private party, wanting to have a property valued. For instance, before a bank can set up a mortgage, it needs to know how much the property for the mortgage is worth, to decide on the size of the mortgage. The *administrative employee* receives and handles new valuation requests, and takes care of planning for the valuers, for whom specific valuations are booked at specific times. The *valuer* does the actual work of valuing the property. The *manager* overlooks all that happens and sometimes performs some work himself.

During the process, at certain times any person fulfilling a role can perform an activity. At other times, for instance after a valuation has been booked, the workflow will demand the specifically selected valuer to perform the activity of valuing the property.

7.2.2 Workflow Overview

Figure 7.1 gives a basic overview of the valuation process. Not all detailed sub-activities are mentioned. We will now shortly clarify the different steps in the process.

The workflow is initiated by a *client*, requesting a valuation for a certain property. This request is entered into the system by the *administrative employee*. This action will result in a new valuation request, having the status ‘Request Received’. Later, the administrative employee makes a booking for a specific valuer, to perform the valuation at a specific time, which will cause the status to change from ‘Request Received’ to ‘Booked’.

The *valuer* will perform the valuation, during which he fills in multiple forms related to different fields of interest connected with the property. During the time he has entered some information, but not all, the request status will be ‘Pending’. After all information is collected, and conclusions are drawn, the valuer *finalizes* the valuation, which will cause the status to continue to ‘Awaiting Approval’.

The *manager* checks all valuations before the results are sent to the requesting client. This means that he looks at valuation requests with status ‘Awaiting Approval’, and approves them if they are good. This will cause the status to switch to ‘Approved’.

From the results of an approved valuation, a PDF report can be generated including an invoice with a price based on information related to the valuation request, created by the administrative employee. This report can be sent to the requesting client. This is done by the *administrative employee*. Sending the results will cause the status to switch to ‘Sent’.

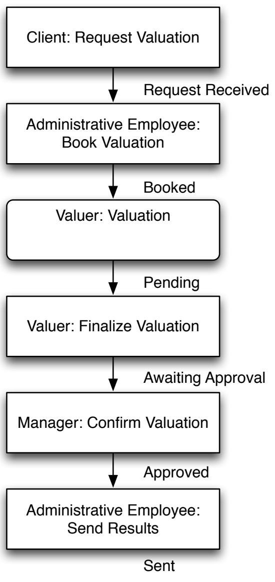

Client: Request Valuation
Request Received
Administrative Employee: Book Valuation
Booked
Valuer: Valuation
Pending
Valuer: Finalize Valuation
Awaiting Approval
Manager: Confirm Valuation
Approved
Administrative Employee: Send Results
Sent

Figure 7.1: Valuation Workflow

7.2.3 Current Implementation

Currently, the working implementation of the valuation system consists of several parts: a *desktop application* connected to a database, a *PDA application* for performing valuations on location, and an *online valuation application* for doing valuations using a web page.

The *desktop application* is a non-workflow enabled application that takes care of the whole business administration. This includes setting privileges for users, changing field values for the valuation application and also some accounting functionality. It is developed using .NET on a Windows platform, and the connected database is a MySQL database. The valuation workflow is, in fact, an ad hoc workflow system implemented in .NET.

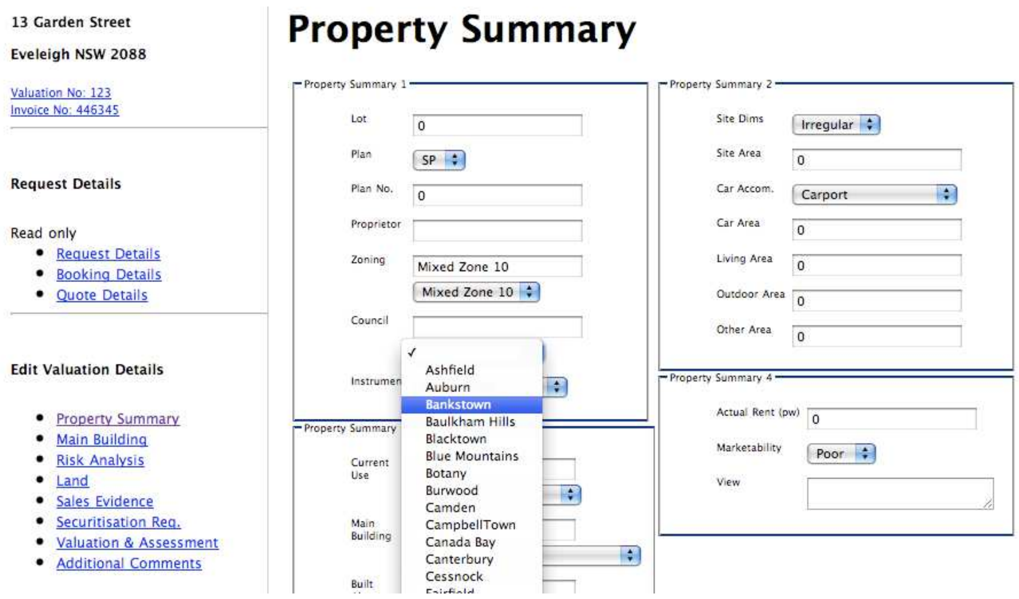
The *PDA application* is a .NET Windows application that allows valuers to take their PDA to a property, input the necessary data using the PDA, and send a finalized valuation report to the server. Communication between the PDA and the database backend is handled using web services.

Afterwards, the *online valuation application* was developed by NICTA, to allow short-term contracted valuers to do their valuation using the internet, because they do not have a PDA. This application was developed using Java Servlets, and includes an integrated valuation workflow system developed in Java. The server alters the data in the central database. While the desktop program includes all administrative applications, the online valuation application is solely for performing the actual valuation and approval by the manager. The valuer fills in several forms with valuation data, after which he finalizes the valuation. Next, the manager has a new task to approve the valuation. A PDF valuation report can also be generated.

7.3 WebWorkFlow Implementation of Valuation

In this section, we show our implementation of the valuation workflow. Whereas the current online valuation application is only concerned with the actual valuation and its approval, we have decided to expand the borders of our application to the whole process, from adding a new valuation request to approving the valuation. However, we do not generate a PDF report from it, as this is not supported in WebDSL.

In the remainder of this section, we will first give a general overview of the valuation application. After this, we will illustrate the data model. We will explain some choices concerning parallellism we had to make in the course of designing the workflow, and show the final result of the workflow.



13 Garden Street
Eveleigh NSW 2088

Valuation No: 123
Invoice No: 446345

Request Details

Read only
- Request Details
- Booking Details
- Quote Details

Edit Valuation Details

- Property Summary
- Main Building
- Risk Analysis
- Land
- Sales Evidence
- Securitisation Req.
- Valuation & Assessment
- Additional Comments

Property Summary

Property Summary 1

Lot	0
Plan	SP
Plan No.	0
Proprietor	
Zoning	Mixed Zone 10
	Mixed Zone 10
Council	
Instrumen	

Property Summary 2

Site Dims	Irregular
Site Area	0
Car Accom.	Carport
Car Area	0
Living Area	0
Outdoor Area	0
Other Area	0

Property Summary 4

Actual Rent (pw)	0
Marketability	Poor
View	

Property Summary

Current Use

Main Building

Built

Figure 7.2: Valuation Form

7.3.1 Valuation Application Overview

We have taken the following approach to implementing the workflow. First, we developed a non-workflow-enabled version of the application, consisting of an application with the possibility to create and alter valuation requests, and with the editing screens that also appear in the current workflow implementation. Next, we added the actual workflow to guide the process of performing all steps like booking a valuation, filling in the different information forms and finalizing valuations, so we could experience implementing the workflow in isolation from the tedious task of designing editing forms.

While implementing the valuation workflow, we noticed that most of the development time was spent doing the following things: defining a data model including all necessary value types, and constructing the forms for entering data. The valuation forms, which are manifold and contain much data, are especially labour-intensive. An example of a constructed page can be seen in figure 7.2. As is visible under the 'Edit Valuation Details' heading on the left side of the screenshot, there are eight different valuation detail forms like this. Defining the necessary value types initially was a lot of work. After we introduced the notion of `string-select-entities` however, it needed much less work.

After creating the data model and designing all edit forms for the valuation request and the valuation itself, it was but a small step to add a workflow to guide the process. This was done by first changing the edit forms to procedure pages, and subsequently constructing a workflow that combined the different procedures in a process definition. We will later show the workflow and the considerations that led us to its design.

7.3.2 Data Model

A simplified data model is displayed in figure 7.3. The central entity in the data model is the `ValuationRequest`. It has numerous properties for containing the valuation information collected during the process. Besides these properties, it has a reference to a `Broker`, a `Client`, a `Valuer`, and possibly multiple `Invoices`. A valuer always has a matching `User` containing login information for the valuer. Users are also connected to `Authorizations`, for giving a user specific authorizations for booking or approving a valuation.

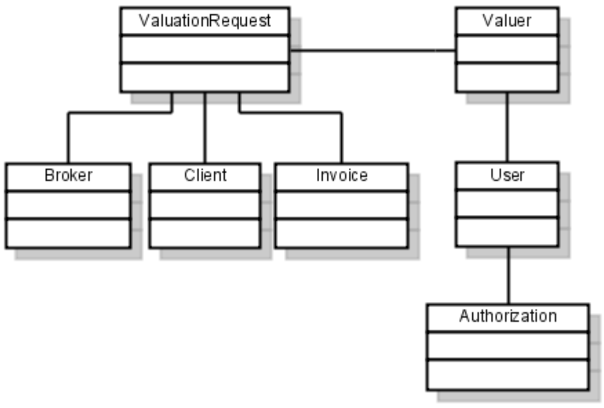


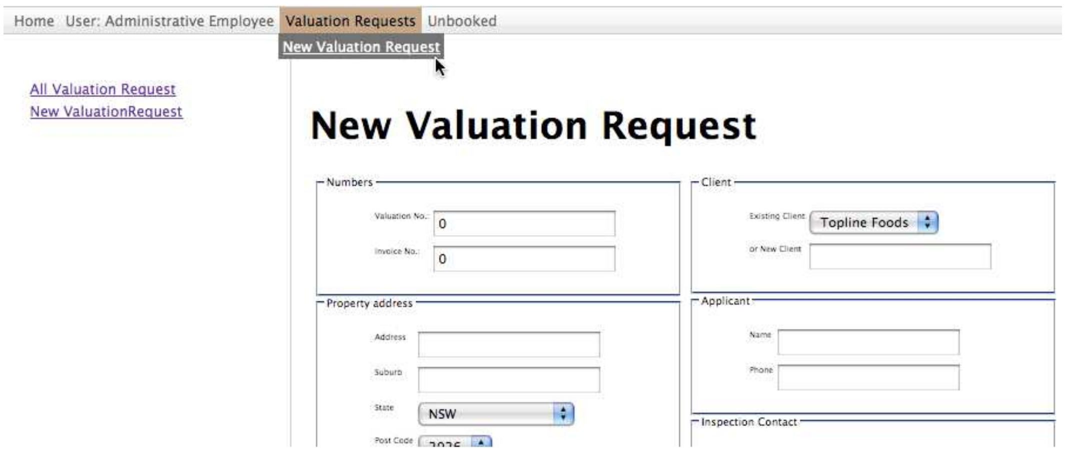
Figure 7.3: Valuation Data Model

This data model does not display the actual properties of the entities. Neither does it display any of the workflow-related entities with which it is later augmented for keeping track of the process.

7.3.3 Workflow

We have implemented the workflow using a process definition in which all procedures are combined. In this section, we show and explain the final result of our workflow design. The workflow starts by creating a valuation request object, done by the administrative employee, as shown in figure 7.4.

The workflow is executed according the following process definition:



New Valuation Request

Numbers

Valuation No.: 0

Invoice No.: 0

Client

Existing Client: Topline Foods

or New Client

Property address

Address

Suburb

State: NSW

Post Code: 2026

Applicant

Name

Phone

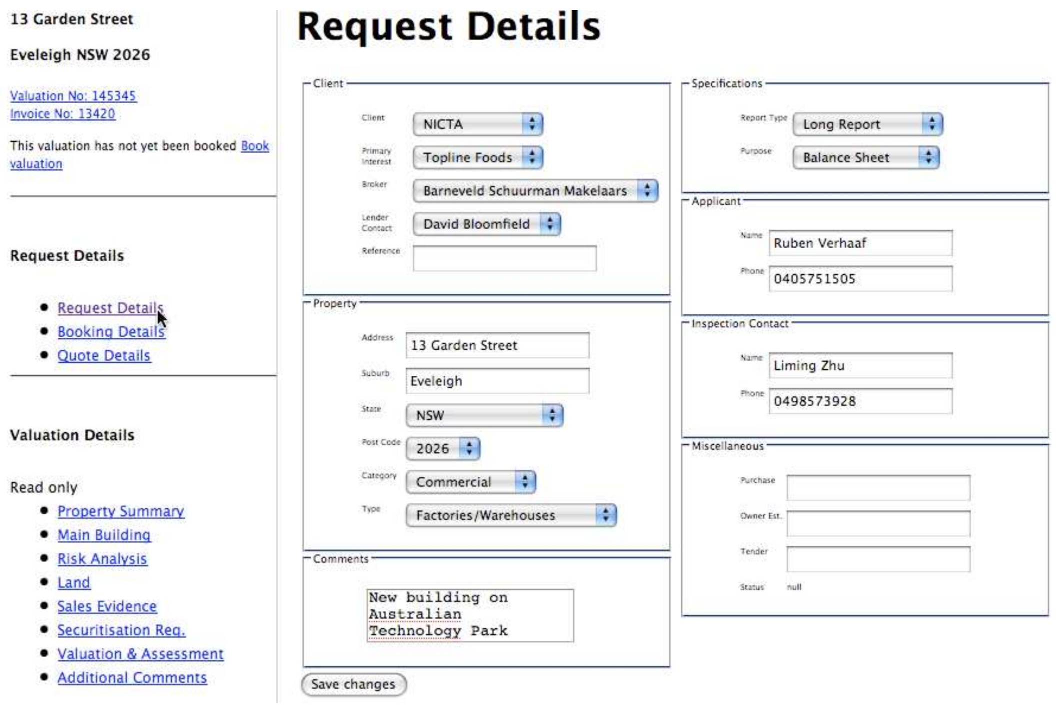
Inspection Contact

Figure 7.4: New Valuation Request

```
auto procedure valuationWorkflow(v : ValuationRequest) {
  process {
    enableEditRequest(v)
    ; bookValuation(v)
    ; repeat {
        editValuationScreens(v)
      } until finalizeValuation(v)
    ; while (true) {
        editValuationScreens(v)
      }
      xor
      approveValuation(v)
    ; disableEditRequest(v)
    ; sendValuation(v)
  }
}

auto procedure editValuationScreens(v : ValuationRequest) {
  process {
    editValuationProperty(v)
    xor editValuationMainBuilding(v)
    xor editValuationRisk(v)
    xor editValuationLand(v)
    xor editValuationSales(v)
    xor editValuationSecuritisation(v)
    xor editValuationAssessment(v)
    xor editValuationComments(v)
  }
}
```

The process starts with a procedure that enables the procedures used for editing a request. This is done asynchronously, using procedure-level amendments to the process definition, to make these procedures available throughout the course of the process. The enabled procedures are only disabled just before sending the valuation, to continuously allow for changes in the request-related information. This can be seen in figure 7.5.



13 Garden Street

Eveleigh NSW 2026

Valuation No: 145345
Invoice No: 13420

This valuation has not yet been booked Book valuation

Request Details

- Request Details
- Booking Details
- Quote Details

Valuation Details

Read only

- Property Summary
- Main Building
- Risk Analysis
- Land
- Sales Evidence
- Securitisation Req.
- Valuation & Assessment
- Additional Comments

Request Details

Client

Client	NICTA
Primary Interest	Topline Foods
Broker	Barneveld Schuurman Makelaars
Lender Contact	David Bloomfield
Reference	

Property

Address	13 Garden Street
Suburb	Eveleigh
State	NSW
Post Code	2026
Category	Commercial
Type	Factories/Warehouses

Comments

```
New building on
Australian
Technology Park
```

Specifications

Report Type	Long Report
Purpose	Balance Sheet

Applicant

Name	Ruben Verhaaf
Phone	0405751505

Inspection Contact

Name	Liming Zhu
Phone	0498573928

Miscellaneous

Purchase	
Owner Est.	
Tender	
Status	null

Figure 7.5: Edit Valuation Request

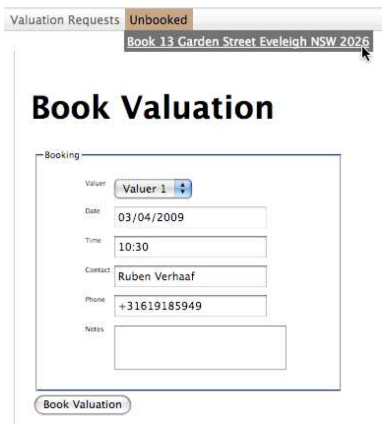
In figure 7.6, the view page for a request is displayed. This page can also be viewed by actors that are not an administrative employee.

Both `enableEditRequest` and `disableEditRequest` are defined as automated procedures, to make their execution invisible for the user:

```
auto procedure enableEditRequest(v : ValuationRequest) {
  do { v.startEditValuationRequestDetails();
       v.startEditValuationRequestBooking();
       v.startEditValuationRequestQuote();   } }

auto procedure disableEditRequest(v : ValuationRequest) {
  do { v.editValuationRequestDetails.disable();
       v.editValuationRequestBooking.disable();
       v.editValuationRequestQuote.disable();   } }
```

After the request procedures are enabled, the valuation is *booked*, as in figure 7.7, which means it is scheduled to be performed by one of the available valuers. The

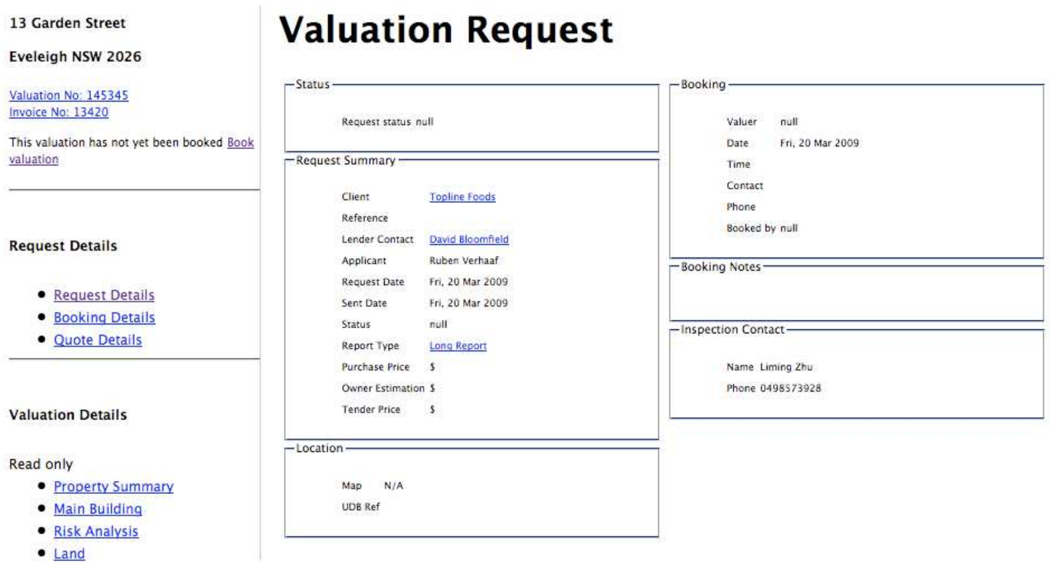


Book Valuation

Booking

Valuer	Valuer 1
Date	03/04/2009
Time	10:30
Contact	Ruben Verhaaf
Phone	+31619185949
Notes	

Figure 7.7: Book Valuation


13 Garden Street

Eveleigh NSW 2026

Valuation No: 145345
Invoice No: 13420

This valuation has not yet been booked Book valuation

Request Details

- Request Details
- Booking Details
- Quote Details

Valuation Details

Read only
- Property Summary
- Main Building
- Risk Analysis
- Land

Valuation Request

Status

Request status null

Request Summary

Client	Topline Foods
Reference	
Lender Contact	David Bloomfield
Applicant	Ruben Verhaaf
Request Date	Fri, 20 Mar 2009
Sent Date	Fri, 20 Mar 2009
Status	null
Report Type	Long Report
Purchase Price	$
Owner Estimation	$
Tender Price	$

Location

Map	N/A
UDB Ref	

Booking

Valuer	null
Date	Fri, 20 Mar 2009
Time	
Contact	
Phone	
Booked by	null

Booking Notes

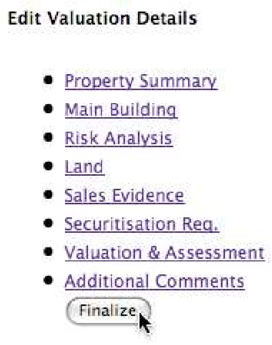
Inspection Contact

Name	Liming Zhu
Phone	0498573928

Figure 7.6: View Valuation Request

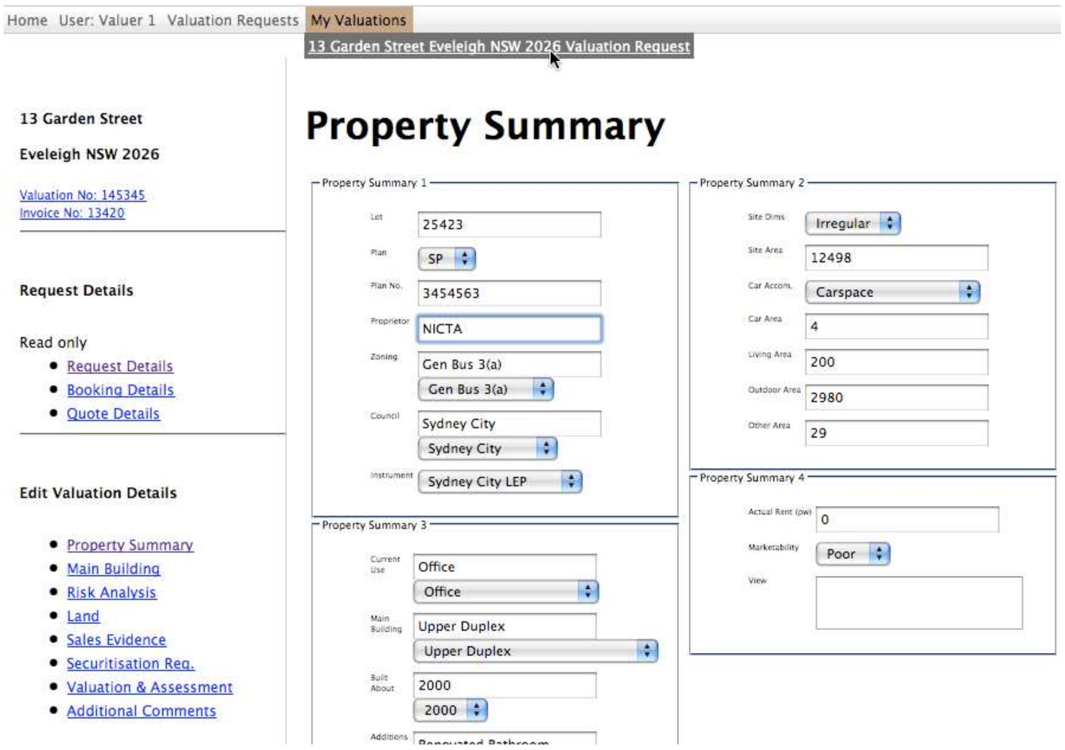
appointed valuer, henceforth, is able to edit all parts of the valuation. He remains able to do this until he *finalizes* the valuation (figure 7.8). After this, the next stage in the process commences: approving the valuation. During the time the `approveValuation` procedure is available to the manager, he can also repeatedly perform amendments to the valuation using the same procedures the valuer used during the previous stage.

For both the *booked* state and the *awaiting approval* state, the same set of procedures is used to edit the valuation data. These procedures are joined using `xor` constructs, and grouped separately using the procedure `editValuationScreens`. To make sure that when the valuation has the status *awaiting approval*, only the designated users can edit the valuation fields, the function `canEditValuation(v : ValuationRequest)` is used in the access control clause. This will cause any user that has authorization for editing valuations to be able to change the valuations both at the moment the valuation is performed, and when it is awaiting approval. A user without specific valuation authorization, however, can only edit the valuation if he is the valuation's valuer, and the valuation status is *booked*. If a user only has authorization for editing valuations that are *awaiting approval*, he will only be able to edit the valuation at that stage of the workflow.


Edit Valuation Details

- Property Summary
- Main Building
- Risk Analysis
- Land
- Sales Evidence
- Securitisation Req.
- Valuation & Assessment
- Additional Comments

Figure 7.8: Finalize Valuation



13 Garden Street

Eveleigh NSW 2026

Valuation No: 145345
Invoice No: 13420

Request Details

Read only

- Request Details
- Booking Details
- Quote Details

Edit Valuation Details

- Property Summary
- Main Building
- Risk Analysis
- Land
- Sales Evidence
- Securitisation Req.
- Valuation & Assessment
- Additional Comments

Property Summary

Property Summary 1

Lot	25423
Plan	SP
Plan No.	3454563
Proprietor	NICTA
Zoning	Gen Bus 3(a)
	Gen Bus 3(a)
Council	Sydney City
	Sydney City
Instrument	Sydney City LEP

Property Summary 2

Site Dims	Irregular
Site Area	12498
Car Accom.	Carspace
Car Area	4
Living Area	200
Outdoor Area	2980
Other Area	29

Property Summary 3

Current Use	Office
	Office
Main Building	Upper Duplex
	Upper Duplex
Built About	2000
	2000
Additions	

Property Summary 4

Actual Rent (pw)	0
Marketability	Poor
View	

Figure 7.9: Valuation

```
// for editing all valuations, authorization is needed
function canEditAllValuations() : Bool {
  for (aut : Authorization where aut.right == "editValuations") {
    if (aut.user == securityContext.principal) {
      return true;
    }
  }
  return false; }

// valuers can only edit their own valuations
function canEditValuation(v : ValuationRequest) : Bool {
  return (
    canEditAllValuations() ||
    (v.status != null && v.status.name == "Booked" &&
     v.valuer != null && v.valuer.user == securityContext.principal) ||
    (v.status != null && v.status.name == "Awaiting Approval" && canEditApprovalValuations())
  ); }
```

After the valuation is approved as in figure 7.10, editing the request fields is disabled. The process ends with sending the valuation results to the requestor, which is normally done by the administrative employee as shown in figure 7.11.

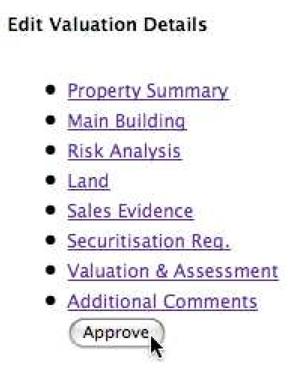


Figure 7.10: Approve Valuation

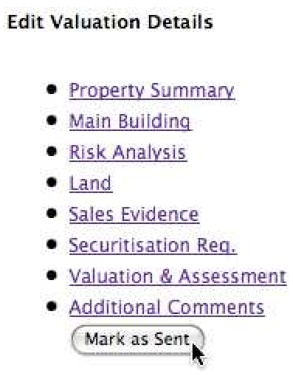


Figure 7.11: Mark as Sent

Procedures

As usual, the process expression merely describes the *control-flow* of the workflow. Conditions for performing the procedures, and actors that are allowed to perform the procedures, are given at the procedure's definition. The following code displays the procedure definitions.

```
procedure bookValuation(v : ValuationRequest) {
  who  { securityContext.principal != null && securityContext.principal.hasBookingRights() }
  view {
    main()
    define local body() {
      header{text("Book Valuation")}
      form {
        table {
          row {
            block("datawidth") {
              group("Booking") {
                groupitem { label("Valuer") { input(v.valuer) } }
                groupitem { label("Date") { input(v.bookingDate) } }
                groupitem { label("Time") { input(v.bookingTime) } }
                groupitem { label("Contact"){ input(v.bookingContact) } }
                groupitem { label("Phone"){ input(v.bookingPhone) } }
                groupitem { label("Notes") { input(v.bookingNotes) } }
              }
            }
            row { action("Book Valuation", do()) }
          }
        }
      }
    }
  }
  do   {
    v.status := initValuationRequestStatus2;
    v.bookedBy := securityContext.principal;
    v.persist();
  } }

procedure editValuationRequestDetails(v : ValuationRequest) {
  who  { canEditValuationRequest(v) }
  view {
    main()
    define sidebar() {
      valuationRequestSidebar(v)
    }
    define body() {
      header{text("Request Details")}
      form {
        ...
      }
    }
  }
  do   { v.persist(); }
  processed { v.editValuationRequestDetails.enable(); } }

procedure editValuationRequestBooking(v : ValuationRequest) {
  who  { canEditValuationRequest(v) }
  view { ... }
  do   { v.persist(); }
  processed { v.editValuationRequestBooking.enable(); } }

procedure editValuationRequestQuote(v : ValuationRequest) {
  who  { canEditValuationRequest(v) }
  view { ... }
  do   { v.persist(); }
  processed { v.editValuationRequestQuote.enable(); } }

procedure editValuationProperty(v : ValuationRequest) {
  who  { canEditValuation(v) }
  view { ... }
  do   { v.persist(); } }

procedure finalizeValuation(v : ValuationRequest) {
  who  { canEditValuation(v) }
  when { v.editValuationProperty.isProcessed
         && v.editValuationMainBuilding.isProcessed
         && v.editValuationRisk.isProcessed
         && v.editValuationLand.isProcessed
         && v.editValuationSales.isProcessed
         && v.editValuationSecuritisation.isProcessed
         && v.editValuationAssessment.isProcessed
         && v.editValuationComments.isProcessed }
  do   { // set status to "Awaiting Approval"
         v.status := statusAwaitingApproval;
         v.persist(); } }
```

NB: There are 7 other procedures like *editValuationProperty*: *editValuationMainBuilding*, *editValuationRisk*, *editValuationLand*, *editValuationSales*, *editValuationSecuritisation*, *editValuationAssessment*, and *editValuationComments*. These are omitted for the sake of brevity.

In most procedures, only the `who` condition is used. For instance, a valuation can only be booked if the function `hasBookingRights()` returns `true` for the logged in user:

```
entity User {
  ...
  function hasBookingRights() : Bool {
    for (a : Authorization where a.right == "booking" && a.user == this) {
      return true;
    }
    return false;
  }
}
```

Another noteworthy phenomenon is that the three procedures used for changing request-related data, *editValuationRequestDetails*, *editValuationRequestBooking*, and *editValuationRequestQuote*, enable themselves during `processed`. This allows the administrative employee to continually change the request information until these procedures are explicitly disabled at the end of the process.

Additional Conditions

The `finalizeValuation` procedure has an additional condition for execution. The actor that can finalize the valuation is any actor for whom the function call to `canEditValuation(v)` returns `true`. In other words, the same users that can edit valuation details at this stage. An additional condition is given in the `when` clause. Without this condition, the `finalizeValuation` procedure would be available once it is enabled, which is after any of the `editValuation` procedures have been performed. However, we want the valuation to be completed before it can be finalized. Therefore, this condition states that the valuation can only be finalized after performing all `editValuation` procedures.

This also demonstrates the power of procedure-level amendments to the process definition. Using the process language, it is not possible to specify that `finalizeValuation` should only become available after all the procedures for entering valuation data are performed at least once. Using the procedure level definitions, it is straightforward.

The action specified in the procedures changes the status of the valuation request to *awaiting approval*. This status field is used to keep track of what happens in a running process, and to have a clear view on its execution.

7.3.4 Parallellism Issues

As can be seen in the workflow process definition, a procedure-level amendment is used to allow administrative employees to be able to alter the valuation request fields throughout the process, by asynchronously starting the `editRequest` procedures, and disabling them at the end of the process. We could have solved this in a different way, however.

One solution is to include the procedures in a parallel path in the process definition. Changing the process definition as follows would do this.

```
auto procedure valuationWorkflow(v : ValuationRequest) {
  process {
    while (true) {
      editValuationRequestDetails(v)
      xor editValuationRequestBooking(v)
      xor editValuationRequestQuote(v)
    } xor finalizeRequestData(v)
    and (
      bookValuation(v)
      ...
      approveValuation(v)
    )
    ; sendValuation(v) } }
```

This solution would have added a parallel path for editing the valuation request data. This would enable administrative employees to change the valuation request data throughout the process. It has the advantage over the other solution that all control resides with the process definition execution. However, an important difference with the present solution is that there is an explicit step in which the administrative employee has to finalize the request data. This adds an extra action to the administrative employee's responsibilities, that can prevent the workflow from continuing execution after the valuation has been approved. Therefore, this solution can be considered inferior to the solution we have used.

7.4 WebWorkFlow Improvements

The case study has led us to some improvements we made to WebWorkFlow, to add a new concise way for defining value entities for use in data-intensive applications, using `string-select-entity`. Also, we added a new form input type: *string input fields with predefined options*. In this section, we describe these improvements to WebWorkFlow.

7.4.1 string-select-entity

While doing the case study, we noticed that many fields needed a custom data entity definition to create a type presenting several string options for selecting, to which values should be able to be added as well. An example is the field `propertyCategory`, for which the following code is needed:

```
entity PCValue {
  name :: String }

define page pCValue(object : PCValue) {
  derive viewPage from object }

define page newPCValue() {
  var object : PCValue := PCValue{};
  derive createPage from object }
```

```
define page allPCValue() {
  table {
    row { navigate(newPCValue()){ "Add value" } }
    for (object : PCValue) {
      row { navigate(pCValue(object)){ text(object.name) } }
    } } }

init {
  var initValue1 : PCValue := PCValue{name := "Commercial"} ;
  initValue1.persist();
  var initValue2 : PCValue := PCValue{name := "Industrial"} ;
  initValue2.persist();
  var initValue3 : PCValue := PCValue{name := "Miscellaneous"} ;
  initValue3.persist();
  var initValue4 : PCValue := PCValue{name := "Residential"} ;
  initValue4.persist();
  var initValue5 : PCValue := PCValue{name := "Retail"} ;
  initValue5.persist();
  var initValue6 : PCValue := PCValue{name := "Unknown"} ;
  initValue6.persist(); }
```

After defining the entity, three different pages are defined: *propertyCategoryValue*, *newPropertyCategoryValue* and *allPropertyCategoryValue*. Finally, for the 6 different values that are known at design time, initial values are given.

To cut back the amount of boilerplate code that was necessary for defining the approximately 50 different types alike, and to save time during development, we decided to introduce a new construction called *string-select-entity*. Instead of using the code above for defining the type needed, it is now sufficient to use one, much more concise, `string-select-entity` definition:

```
string-select-entity PropertyCategory {
  "Commercial", "Industrial", "Miscellaneous", "Residential", "Retail", "Unknown"
}
```

From this construction, the entities and initial values resulting from the code above are automatically generated. While the original way of defining the entity cost around 35 lines of code, a string-select-entity is generally constructed using only one line, which shows that the boilerplate code is indeed cut back dramatically.

7.4.2 String Input Fields with Predefined Options

During the course of designing the valuation forms, one relatively often used input field type was unavailable in WebDSL: a textbox for string input with an associated drop-down list to allow selecting a predefined string option, that copies the selected string value into the textbox to make entering recurring values easier. We decided to add a feature to WebDSL to enable this input field type. By adding syntax for a property annotation when defining entities, we were able to add the possibility to connect a `string` field with a `string-select-entity` field, as illustrated by the following example:

```
entity ValuationRequest {
  ...
  council :: String (select=CouncilOption)
  ...
}

string-select-entity CouncilOption{"Ashfield", "Auburn", "Bankstown", ...}
```

This code extract results in adding a `string` field to `ValuationRequest`, that when displaying in an input form, will result in the display of two different input fields as described before. See figure 7.2 for a screenshot of a working version of this field input.

7.5 Case Study Results

In this section, we will shortly evaluate the results of the case study performed in this chapter. We will do this by first examining the amount of code that goes into defining the workflow, and comparing this to the rest of the code. After this, we will mention several deficiencies of WebWorkFlow, together with the possible improvements that could be made. We end with distinguishing several benefits of the use of WebWorkFlow for workflows like the LIXI valuation workflow.

7.5.1 Workflow Code Size

To evaluate the use of WebWorkFlow, we will look at the lines of code needed for different parts of the valuation system, as well as the amount of bytes used by the code. The amount of lines of code can depend on the style of programming, but the amount of bytes needed is a less volatile measure. An overview of the amount of code needed is given in figure 7.12.

The results show that the valuation system needs about 2500 lines of code. The workflow part of that is 230 lines, plus about 170 lines of access control functions hidden in *application code*, which amounts to a total of 400 lines: only 16% of the total program size. Only 30 lines of code are used for specifying the control-flow of the process to provide scheduling for the procedures, and 200 lines are used for defining the procedures themselves.

If we compare the size in bytes, we see that the results are fairly similar. Of a total of about 83000 bytes, only about 9200 bytes are used for the workflow part: 11% of the total code size.

The rest of the code can be split into a *data model*, and code for the *user interface*. Of these parts, the user interface part is clearly the largest: close to two-third of the application consists of user interface code. This user interface code consists of page definitions, templates and code for style and layout. Page definitions are the raw definitions for how the different pages for viewing and editing valuations should look. Templates are used to provide the pages with a standardized layout and navigation menus.

	LOC	**%**	**Bytes**	**%**
Process description	30	1%	700	1%
Procedure definitions	200	8%	4000	5%
Application code	270	11%	7100	9%
Data model	375	15%	18600	22%
User interface	1600	65%	53000	64%
	2475	100%	83400	100%

Figure 7.12: Code Size (WebWorkFlow)

Style and layout is used for the markup of these pages. The data model used for the valuation system is extensive, because property valuations consist of a large amount of information. Nonetheless, the data model comprises only 15% of the application code. However, in terms of bytes of code, this figure is slightly larger. Due to a very efficient method for specifying lists of string data, 22% of the code is contained in these 15% of the lines of code.

With only 9 percent of the lines of code, and only 6% of the amount of bytes dedicated to the actual procedure and process definitions, we can safely say that WebWorkFlow provides a very concise way of specifying a workflow.

7.5.2 Code Size Comparison

When it comes to measuring the degree to which WebWorkFlow reduces boilerplate code, the best measurement is a comparison of our implementation to the original implementation. We have inspected the code of the original application and summarized the results measured in lines of code and bytes of code as visible in figure 7.13.

A noteworthy fact is that the original application only contains code for supporting the actual valuation and approving the results, while the WebWorkFlow implementation supports the whole process, starting with the request to perform a property valuation.

	LOC	**%**	**Bytes**	**%**
Application code	4600	30%	138800	33%
Data model	7800	51%	210600	49%
User interface	3000	19%	77000	18%
	15400	100%	426400	100%

Figure 7.13: Code Size (Original Implementation)

In these figures, we have not counted the code used for making the PDF reports, as our implementation does not support creating PDF reports. Including this in the comparison would not be fair. In figure 7.14, a comparison between the two implementations is made. The last column in the table signifies the factor of code length and space gained by using WebWorkFlow.

The total amount of lines of code of which the original implementation consists is 15,400. This is roughly 6 times more than the amount of code used in our WebWorkFlow implementation, despite the fact that the WebWorkFlow implementation of valuation includes the whole process starting with receiving the request, and not just the actual valuation and approval. This is a substantial improvement. In terms of bytes of code, the reduction is a factor of 5.

Lines of Code

	Original	**WebWorkFlow**	**Factor**
Application code	4600	500	9.2
Data model	7800	375	20.8
User interface	3000	1600	1.9
Total	**15400**	**2475**	**6.2**

Bytes

	Original	**WebWorkFlow**	**Factor**
Application code	138800	11800	11.8
Data model	210600	18600	11.3
User interface	77000	53000	1.5
Total	**426400**	**83400**	**5.1**

Figure 7.14: Code Reduction

It is also interesting to compare the reduction factors of different parts of the application. If we look at the byte count, we see that the reduction of both the application code and the data model are between a factor 11 and a factor 12. The user interface, however, only sees a reduction of a factor 1.5. The amount of lines of code differs from this measure: reduction in application code is a factor 9, reduction of the UI code is a factor 2 and reduction of the data model is as high as a factor 21.

Thus, we think that WebWorkFlow does exceptionally well when it comes to reducing data model code. Also, there is a substantial reduction in application code. On the other hand, the user interface code reduction of the WebWorkFlow implementation is much less. We think there are several reasons for this:

- In the original implementation, 33% of the code size is dedicated to the functioning of the main application. WebWorkFlow on the other hand, does not need any auxilary code for its user interface to work properly. Therefore, the user interface is a larger part of the application, incorporating much of the logic that would otherwise be specified in other parts of the application code.
- There is less user interface boiler plate code that can be cut. Therefore, the advantage of WebWorkFlow in specifying user interfaces is smaller than in specifying the rest of the application.
- We can still improve on the user interface language constructs, to cut more boilerplate code.

7.5.3 WebWorkFlow Deficiencies

Doing the case study, we encountered a real-world situation to be modeled using WebWorkFlow. This has made us aware of several aspects of WebWorkFlow that would greatly benefit from improvement. We mention *separation of UI from procedures*, *limits for CSCW*, *input validation*, *absence of web service support*, *no optional values*, and *absence of PDF generation facilities*.

Separating UI from Procedures

While doing the case study, we discovered that there is a small problem with procedures that makes it impossible to separate UI code using templates if used within procedures. The cause of this problem is that when procedures are specified, it is possible to use a call to an action `do()`.

A solution to this problem can be implemented as follows. Instead of adding the procedure action as a function called `doAction()`, and rewriting the `do()` call in a view page to `doAction()`, as is done now, we can add the function as part of a very small template `do(caption : String)` that prints a button with the associated action, and to which we pass the text that should be printed on the button. So instead of

```
row { action("Book Valuation", do()) }
```

for outputting a button for the action, we call the action template.

```
row { do("Book Valuation") }
```

Because templates in WebDSL can be defined in the scope of a page, this does not present problems when generating `do()` templates for multiple procedures: these templates are defined in the scope of the associated procedures page and are thus separated.

Limits for CSCW

When it comes to Computer-Supported Cooperative Work (CSCW), the WebWorkFlow process language has its limits. CSCW applications generally are less rigid and more flexible than most workflow applications. For applications in which the workflow must be flexible, the WebWorkFlow process language is less appropriate.

The Valuation process adheres to this to some extent. Property valuation follows a relatively simple process in which loops are only used to continuously allow changes to valuation data, and the main part of the application is data entry forms. In this case, the execution scheduling of the different procedures might just as easily have been implemented using only procedures. Using only the status indicators that are used anyway, it is possible to devise relatively simple conditions for viewing procedures, instead of using the generated conditions when using the process language.

Limitations of Generated Task Lists

When procedures are specified for entities, any available procedures show up in two different places: in the sidebar of the object's view page, and in the *All Tasks* list. However, for applications in which workflow is used, one typically wants more than an unstructured list of the tasks that can be performed at that moment.

In our case study, we have not used the automatically generated task lists. Instead, we have defined our own sidebar, viewing any available procedures. This costs a considerable amount of code, but it is a necessity for good usability of the application.

A drawback of the approach in which currently available procedures are displayed, is that there is no clear overview over the whole of the process. As procedures that are been performed will disappear afterwards, there is no view on the history of task execution. Also, procedures that will become available in the near future are not displayed, so a roadmap of what lies ahead is not available, either. A last issue is that only procedures that can be performed by the currenly logged in user are displayed. There currently is no way of distinguishing the situation in which the workflow has ended from the situation in which the workflow is waiting for a procedure to be performed, using the generated task lists.

Ideally, one is able to view the complete state of the process, including previously performed procedures, procedures that will be enabled later, and procedures that can be performed by other actors. Displaying procedures that can be performed by other actors is relatively simple, as it amounts to also including procedures in the list for which the `when` condition holds, but the `who` condition does not necessarily hold. Displaying previously performed procedures is slightly more complicated, but can be done by keeping account of procedures that are performed. A more complex task is also including procedures that are not yet enabled, but will be enabled in the future. As one can use arbitrary conditions in WebWorkFlow procedures, a full analysis of all defined procedures is necessary, to create a tree of events that may lead to availability of a procedure in the future, if mainly procedures are used without handling the control-flow with a process description. However, this task is much simpler if a process description is used, because all procedures that might become available later on, are known beforehand.

Also, one might want a slightly more structured way of presenting a list of procedures. Instead of displaying just a list of procedures, it would be informative to for instance display loops as a separate portion of the workflow with a clear indication of the fact that a loop is still active, or has fully completed.

Input Validation

Applications developed using WebDSL are often data-intensive. These applications naturally involve many forms that must be filled in. In general, forms should be validated before saving any contents to the database, or doing any other processing of the data. At the time of performing the case study, this was not handled well in WebDSL. It was possible to do input valuation, but the best solution conceivable was writing explicit checks for each input field, and referring the user to a different page instead of committing the data, if any errors are found.

This is a very elaborate way of providing validation. In the meantime, WebWorkFlow has been altered to incorporate more sophisticated validation that is derived from the data model, so validation can be done automatically. This is done by adding syntax for specifying validation constraints in entities, using attribute annotations.

Absence of Web Service Support

Many workflow solutions use web services for communication between different parties, or computers. BPEL is even built using mainly webservice communication for executing the workflow control-flow, as was explained in greater detail in section 9.5.

In the current version of the valuation workflow implementation, web services are also used to communicate with the server, in the case of performing tasks using a PDA (see section 7.2.3). This connection with an existing workflow system, or part thereof, is impossible using WebWorkFlow.

These facts are enough to conclude that WebWorkFlow would greatly benefit of the addition of web service support to WebDSL.

No Optional Values

There is a problem with the types provided in WebDSL's data model. It is not yet possible to specify that an attribute's value is optional. For example, this results in zeros being inserted into integer fields, when not filled in. This is a serious disadvantage for an application like the valuation workflow, in which large amounts of data fields can be filled in, of which many are optional.

Absence of PDF Generation Facilities

The current version of the valuation workflow implementation has a function that allows the user to generate a PDF report from the valuation data. Unfortunately, it is not yet possible to implement this using WebWorkFlow. This deficiency is a good opportunity for future work, as many data-intensive applications can greatly benefit from PDF generation.

7.5.4 WebWorkFlow Benefits

Process Readability

It is important that the process specification of workflows is readable and provides a clear view on the process. In our experience, this is the case with WebWorkFlow. The process description shown in section 7.3.3 indeed provides a clear bird's eye view on the process. However, for the details of procedure execution, the conditions stated at the definition of procedures are also important. In case one wants to have a more detailed view of the process therefore, one has to also look at the procedure definition code.

Flexible Process Specification

The use of the process and procedure language proves the flexibility of WebWorkFlow, as patterns that are not directly supported by the process language can be constructed using a customization on the procedure level. The abstractions in the WebWorkFlow language extension thus clearly provide a flexible way of defining control-flow for the workflow. We will give two examples of modifications on the procedure level that were used to change the flow of control.

When defining the flow of control, we used the `repeat {} until` construct to make sure that at least one of the different valuation procedures is performed before the valuation is finalized. At the process level, there was no possibility to specify that all valuation procedures had to be performed before the *finalizeValuation* procedure could be performed. At the procedure level, however, only one simple condition was needed to enforce this constraint: by adding a `when` condition to *finalize-Valuation*, we made sure that the valuation cannot be finalized before all valuation procedures have been performed. See the code snippet in section 7.3 for an example.

The second example is the example of valuation request edit pages. To enable the asynchronous enabling of procedures for editing request pages, procedure-level code was used to extend the process expression's functionality, using two separate procedures to enable and disable the procedures: `enableEditRequest` and `disableEditRequest`.

Chapter 8

Workflow Patterns Coverage

In this chapter, we evaluate WebWorkFlow on the basis of the coverage in terms of workflow patterns, as formulated by Van der Aalst et al. [61, 55, 52, 53]. We will discuss all three different workflow perspectives described in section 9.3, namely *control-flow*, *resource* and *data*.

8.1 Control-Flow Patterns

Pattern 1	Sequence	supported
Pattern 2	Parallel split	supported
Pattern 3	Synchronization	supported
Pattern 4	Exclusive choice	supported
Pattern 5	Simple merge	supported
Pattern 6	Multi-choice	partially supported
Pattern 7	Synchronizing merge	supported
Pattern 8	Multi-merge	partially supported
Pattern 9	Discriminator	supported
Pattern 10	Arbitrary cycles	supported
Pattern 11	Implicit termination	unsupported
Pattern 12	Multiple instances without synchronization	partially supported
Pattern 13	Multiple instances with a priori design time knowledge	partially supported
Pattern 14	Multiple instances with a priori runtime knowledge	partially supported
Pattern 15	Multiple instances without a priori runtime knowledge	partially supported
Pattern 16	Deferred choice	supported
Pattern 17	Interleaved parallel routing	unsupported
Pattern 18	Milestone	supported
Pattern 19	Cancel activity	partially supported
Pattern 20	Cancel case	partially supported

Figure 8.1: Control-flow Pattern Coverage

In this section, we will explore WebWorkFlow's coverage of the control-flow patterns described in [61]. A brief overview of the results can be viewed in the table above. The table in figure 8.1 shows that several patterns are partially supported. When this is mentioned, it is mostly in the case of patterns involving multiple instances, where we had to resolve to using a dummy entity to represent an instance.

8.1.1 Pattern 1 - Sequence

Description: An activity in a workflow process is enabled after the completion of another activity in the same process.

Sequence of procedures is naturally supported using WebWorkFlow's sequential operator `;`.

8.1.2 Pattern 2 - Parallel split

Description: A point in the workflow process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order.

This is supported in WebWorkFlow using the `and` operator, as in the following example:

```
process {
  employeeFillInForm(p) and managerFillInForm(p)
}
```

8.1.3 Pattern 3 - Synchronization

Description: A point in the workflow process where multiple parallel subprocesses / activities converge into one single thread of control, thus synchronizing multiple threads. It is an assumption of this pattern that each incoming branch of a synchronizer is executed only once.

WebWorkFlow supports this pattern using the `and` operator, as shown in the example for pattern 2: Parallel split.

8.1.4 Pattern 4 - Exclusive choice

Description: A point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen.

This is supported using the `if (condition) {} [else {}]` construct as showed in the following example:

```
process {
  if (p.reportIsFinal) {
    evaluateReport(p)
  } else {
    editReport(p)
  }
}
```

8.1.5 Pattern 5 - Simple merge

Description: A point in the workflow process where two or more alternative branches come together without synchronization. It is an assumption of this pattern that none of the alternative branches is ever executed in parallel.

As the `xor` operator in WebWorkFlow has the meaning that either of the two branches will finish, and the other one will not, one could argue that part of both branches would be executed in parallel. However, the `+` operator in WebWorkFlow has the meaning that once an activity within either of the branches is registered, the other branch will automatically be disabled, and they will in fact not run in parallel. This is demonstrated in the following example:

```
process {
  evaluateReport(p) + editReport(p)
}
```

8.1.6 Pattern 6 - Multi-choice

Description: A point in the workflow process where, based on a decision or workflow control data, a number of branches are chosen.

This is partly supported in the WebWorkFlow process language, as there are no separate split and join constructs. Using an `and` operator or an `xor` operator in combination with two conditional `if (condition)` {} blocks the behaviour can easily be mimicked:

```
process {
  if (p.contributors <= 2) {
    addContributor(p)
  }
  and
  if (p.reportIsFinal) {
    editReport(p)
  }
}
```

If any different ways of joining the branches is needed, one can always use the procedure level to express this:

```
do {
  if (p.contributors <= 2) {
    p.startAddContributor();
  }
  and
  if (p.reportIsFinal) {
    p.startEditReport();
  }
}
```

8.1.7 Pattern 7 - Synchronizing merge

Description: A point in the workflow process where multiple paths converge into one single thread. If more than one path is taken, synchronization of the active threads needs to take place. If only one path is taken, the alternative branches should reconverge without synchronization. It is an assumption of this pattern that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete.

This pattern is naturally supported by WebWorkFlow, using the operators `and`, `xor` and `+`. It is possible to create an expression using these operators. The operator `and` has a higher priority than `xor` and `+`, which have the same priority. Apart from this, they are executed left first.

The following code snippet provides an example of the use of these constructs intermingled, which causes the different branches to be automatically synchronized/merged:

```
process {
  rejectPaper(p)
  xor
  evaluateReport(p) and evaluateReview(p)
  +
  editReport(p)
}
```

In this example, the user will first choose between the two branches of the `+`. Subsequently, either `rejectPaper(p)` is performed or both procedures joined with the `and` operator are. Synchronizing and merging are performed automatically, in this way.

8.1.8 Pattern 8 - Multi-merge

Description: A point in a workflow process where two or more branches reconverge without synchronization. If more than one branch gets activated, possibly concurrently, the activity following the merge is started for every activation of every incoming branch.

Because of the nature of WebWorkFlow, in which an activity cannot have two running cases for the same workflow object, we have to introduce a dummy entity for multiple instances of a task to be available at the same time. For instance, for an entity `Paper` for which a procedure is defined, we define an entity `PaperWorkflowInstance`, for which we define the procedure that should have multiple instances. Using this solution, WebWorkFlow supports this pattern. This solution is also used in patterns 12 to 15 and in pattern 19 and 20.

```
procedure workflowPart1(i : PaperWorkflowInstance) {
  ...
  ; finish(i)
}

procedure workflowPart2(i : PaperWorkflowInstance) {
  ...
```

```
    ; finish(i)
  }

  procedure conferencePart(p : Paper) {
    do {
      p.i1 := newPaperWorkflowInstance();
      p.i2 := newPaperWorkflowInstance();
      p.persist();
    }
    process {
      workflowPart1(p.i1)
      and
      workflowPart2(p.i2)
    }
  }
```

In the implementation of `finish()`, one can refer to `p` by using the inverse link property `i.p` for `PaperWorkflowInstance`.

8.1.9 Pattern 9 - Discriminator

Description: The discriminator is a point in a workflow process 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 ignores them. Once all incoming branches have been triggered, it resets itself so that it can be triggered again (which is important otherwise it could not really be used in the context of a loop).

This is actually what happens when an `xor` operator is used. Except that the other branch is disabled once a branch completes. If the other branch should be able to continue, this is easily modeled by using the procedure level descriptions to enable the join procedure without disabling the other branches still running:

```
  extend entity Person {
    partTwoStarted :: Bool
  }

  procedure one(p : Person) {
    do {
      if (!p.partTwoStarted) {
        p.partTwoStarted := true;
        p.persist();
        p.startPartTwo();
      }
    }
  }
  procedure two(p : Person) {
    do {
      if (!p.partTwoStarted) {
        p.partTwoStarted := true;
        p.persist();
        p.startPartTwo();
      }
    }
  }
  procedure three(p : Person) {}

  auto procedure pattern(p : Person) {
    process {
      one(p) and two(p)
    }
  }
  auto procedure partTwo(p : Person) {
    process {
      three(p)
    }
  }
```

8.1.10 Pattern 10 - Arbitrary cycles

Description: A point in a workflow process where one or more activities can be done repeatedly.

This is possible by using either the `while (condition) {}` construct or the `repeat {} until` construct:

```
process {
  while (p.count <=5) {
    addReviewer(p)
  }
}

process {
  repeat {
    writeReport(p)
    ; approveReport(p)
  } until finalizeReport(p)
}
```

8.1.11 Pattern 11 - Implicit termination

Description: A given subprocess should be terminated when there is nothing else to be done. In other words, there are no active activities in the workflow and no other activity can be made active (and at the same time the workflow is not in deadlock).

As in WebWorkFlow, the process is implemented as a graph of procedures enabling each other, the implicit termination pattern is not supported. Nor is it necessary: in [65], it is mentioned that implicit termination hides design errors because it is not possible to detect deadlocks. Therefore the absence of support for the pattern would force the designer to think about the process better.

8.1.12 Pattern 12 - Multiple instances without synchronization

Description: Within the context of a single case (i.e., workflow instance) multiple instances of an activity can be created, i.e., there is a facility to spawn off new threads of control. Each of these threads of control is independent of other threads. Moreover, there is no need to synchronize these threads.

This pattern is supported. By adding some code at the procedure level to enable other procedures in which part of the workflow is described, any part of a workflow can be ‘spawn off’. An example:

```
procedure workflowPart(i : PaperWorkflowInstance) {
  ...
}

auto procedure createThread(p : Paper) {
  do {
    var i : PaperWorkflowInstance := newPaperWorkflowInstance();
    p.threadAmount := p.threadAmount - 1;
    p.instances.add(i);
    p.persist();
    i.workflowPart.enable();
  }
}

procedure conferencePart(p : Paper) {
  do {
    p.threadAmount := 3;
  }
  process {
    while(p.threadAmount > 0) {
      createThread(p)
    }
  }
}
```

8.1.13 Pattern 13 - Multiple instances with a priori design time knowledge

Description: For one process instance an activity is enabled multiple times. The number of instances of a given activity for a given process instance is known at design time. Once all instances are completed some other activity needs to be started.

By introducing a dummy entity for instances of tasks, WebWorkFlow supports this pattern. How this is done is showed in the following code sample:

```
procedure workflowPart(i : PaperWorkflowInstance) {
  processed {
    var allFinished := true;
    for (instance : PaperWorkflowInstance in i.p.instances) {
      if (!instance.workflowPart.isProcessed) {
        allFinished := false;
      }
    }
    if (allFinished) {
      i.p.conferencePart2.enable();
    }
  }
}

auto procedure createThread(p : Paper) {
  do {
    var i : PaperWorkflowInstance := newPaperWorkflowInstance();
    p.threadAmount := p.threadAmount - 1;
    p.instances.add(i);
    p.persist();
    i.workflowPart.enable();
  }
}

procedure conferencePart1(p : Paper) {
  do {
    p.threadAmount := 3;
  }
  process {
    while(p.threadAmount > 0) {
      createThread(p)
    }
  }
}

procedure conferencePart2(p : Paper) {}
```

8.1.14 Pattern 14 - Multiple instances with a priori runtime knowledge

Description: For one case an activity is enabled multiple times. The number of instances of a given activity for a given case varies and may depend on characteristics of the case or availability of resources, but is known at some stage during runtime, before the instances of that activity have to be created. Once all instances are completed some other activity needs to be started.

The only difference with pattern 13 is that the amount of threads is not yet known at design time. This is not a problem for WebWorkFlow, as can be seen in this code example, that presents an update on pattern 14's example, in which the decision how many threads should be created is left until runtime:

```
procedure conferencePart1(p : Paper) {
  view {
    derive procedurePage from p for (threadAmount)
  }
  process {
    while(p.threadAmount > 0) {
      createThread(p)
    }
  }
}
```

8.1.15 Pattern 15 - Multiple instances without a priori runtime knowledge

Description: For one case an activity is enabled multiple times. The number of instances of a given activity for a given case 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. Once all instances are completed some other activity needs to be started. The difference with Pattern 14 is that even while some of the instances are being executed or already completed, new ones can be created.

This pattern is supported by WebWorkFlow. As an example, we adapt the example of pattern 14 by applying the following changes:

```
// add this procedure
procedure addAnotherThread(p : Person) {
  when {
    p.conferencePart2 == null
    ||
    (p.conferencePart2 != null    &&
     !p.conferencePart2.isEnabled &&
     !p.conferencePart2.isProcessed)
  }
  process {
    createThread(p)
  }
  processed {
    this.enable();
  }
}

// add a processed {} block to conferencePart1
procedure conferencePart1(p : Paper) {
  ...
  processed {
    p.startAddAnotherThread();
  }
}

// for neatness, we disable addAnotherThread when conferencePart2 is enabled
procedure conferencePart2(p : Paper) {
  enabled { if (p.addAnotherThread != null) { p.addAnotherThread.disable(); } }
}
```

After these changes, we can add new threads as long as not all threads have been finished, as that is the moment at which `p.part2` will be created and enabled.

8.1.16 Pattern 16 - Deferred choice

Description: A point in the workflow process where one of several branches is chosen. In contrast to the XOR-split, the choice is not made explicitly (e.g. based on data or a decision) but several alternatives are offered to the environment. However, in contrast to the AND-split, only one of the alternatives is executed. This means that once the environment activates one of the branches the other alternative branches are withdrawn. It is important to note that 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.

This pattern is directly supported by WebWorkFlow, using the + operator, which has exactly the meaning described above. A small example:

```
process {
  evaluateReview(p) + editReport(p)
}
```

8.1.17 Pattern 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 run-time, and no two activities are executed at the same moment (i.e. no two activities are active for the same workflow instance at the same time).

This pattern is not supported by WebWorkFlow. It would be easy to implement if a random number generator was provided, as in this way, the activition (enabling) of tasks (procedures) can be done in the random manner asked for.

8.1.18 Pattern 18 - Milestone

Description: The enabling of an activity depends on the case being in a specied state, i.e. the activity is only enabled if a certain milestone has been reached which did not expire yet. Consider three activities named A, B, and C. Activity A is only enabled if activity B has been executed and C has not been executed yet, i.e. A is not enabled before the execution of B and A is not enabled after the execution of C.

This pattern is supported by WebWorkFlow, by adding constraints to procedure A. In this way, the milestone behaviour of being able to perform A only between B and C is implemented:

```
procedure a(p : Person) {
  when {
    p.b != null && p.b.isProcessed && p.c != null && !p.c.isProcessed
  }
}
procedure b(p : Person) {}
procedure c(p : Person) {}
procedure conferencePart(p : Paper) {
  process {
    a(p) and b(p) and c(p)
  }
}
```

8.1.19 Pattern 19 - Cancel activity

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

This is supported by calling `object.procedurename.disable()`. When this is done, all descending procedures (related to this one through the `process` description) will be disabled as well. An example (see pattern 12 for an implementation of `createThread`):

```
procedure startAll(p : Person) {
  do {
    p.int := 3;
    p.persist();
  }
  process {
    while(p.int > 0) {
      createThread(p)
    }
  }
}
procedure cancelOthers(p : Person) {
  do {
    for(i : PersonWorkflowInstance in p.instances) {
      if (i.workflowPart != null) { i.workflowPart.disable(); }
    }
  }
}
auto procedure pattern(p : Person) {
  process {
    startAll(p); cancelOthers(p)
  }
}
```

8.1.20 Pattern 20 - Cancel case

Description: A case, i.e. workflow instance, is removed completely (i.e., even if parts of the process are instantiated multiple times, all descendants are removed

As there is no real difference between workflows and activities in WebWorkFlow, this pattern is naturally supported as well.

8.2 Resource Patterns

In this section, we will explore WebWorkFlow's coverage of the resource patterns described in [52], by investigating the ability of WebWorkFlow to capture the first 20 patterns. As for the control-flow patterns, we have provided a table with an overview of the results in figure 8.2.

Pattern 1	Direct allocation	supported
Pattern 2	Role-based allocation	supported
Pattern 3	Deferred allocation	supported
Pattern 4	Authorization	supported
Pattern 5	Separation of duties	supported
Pattern 6	Case handling	supported
Pattern 7	Retain familiar	supported
Pattern 8	Capability-based allocation	supported
Pattern 9	History-based allocation	supported
Pattern 10	Organizational allocation	supported
Pattern 11	Automatic execution	supported
Pattern 12	Distribution by offer - single resource	supported
Pattern 13	Distribution by offer - multiple resources	supported
Pattern 14	Distribution by allocation - single resource	supported
Pattern 15	Random Allocation	unsupported
Pattern 16	Round Robin allocation	supported
Pattern 17	Shortest queue	supported
Pattern 18	Early distribution	supported
Pattern 19	Distribution on enablement	supported
Pattern 20	Late distribution	supported

Figure 8.2: Resource Pattern Coverage

8.2.1 Pattern 1 - Direct Allocation

Description: The ability to specify at design time the identity of the resource that will execute a task.

This pattern is naturally supported by WebWorkFlow using the `who {}` construction for procedures:

```
procedure pattern(p : Person) {
  who { securityContext.principal != null && securityContext.principal.isFred() }
}

// the function used in the above procedure
entity User {
  ...
  function isFred() : Bool {
    return (this.realname == "Fred");
  }
}
```

8.2.2 Pattern 2 - Role-Based Allocation

Description: The ability to specify at design time that a task can only be executed by resources which correspond to a given role.

This pattern is supported by WebWorkFlow. In the next example, entity `User` is extended with a function to check group membership for use in the procedure `who` condition.

```
procedure pattern(p : Person) {
  who { securityContext.principal != null && securityContext.principal.belongsToGroup(valuers) }
}

// the function used in the above procedure
entity User {
  ...
  function belongsToGroup(group : Group) : Bool {
    for (role : Group in this.roles) {
      if (role == group) {
        return true;
      }
    }
    return false;
  }
}
```

8.2.3 Pattern 3 - Deferred Allocation

Description: The ability to defer specifying the identity of the resource that will execute a task until runtime.

This pattern is supported by WebWorkFlow. This is showed in the next example, in which it is implemented with two consecutive tasks in which the first is specifying a user that is authorized for performing the next task.

```
extend entity Person {
  performingUser -> User
}
procedure pattern(p : Person) {
  process {
    step1(p); step2(p)
  }
}
procedure step1(p : Person) {
  view {
    derive procedurePage from p for (performingUser)
  }
}
procedure step2(p : Person) {
  who { securityContext.principal != null && securityContext.principal == p.performingUser }
}
```

8.2.4 Pattern 4 - Authorization

Description: The ability to specify the range of resources that are authorized to execute a task.

This pattern is supported by WebWorkFlow. One way to do this, is to assign a task to a group and, before enabling that task, provide the user with the possibility to select group members for that group, as in the following example.

```
extend entity Person {
  allowedGroup -> Group
}

procedure pattern(p : Person) {
  view {
    form {
      var group : Group := Group {};
      input(group.users)
      action("Save group", do())
    }
  }
  do {
    group.persist();
    p.allowedGroup := group;
    p.persist();
  }
  process {
    step2(p)
  }
}
procedure step2(p : Person) {
  who { securityContext.principal != null && securityContext.principal.belongsToGroup(p.allowedGroup) }
}
```

8.2.5 Pattern 5 - Separation of Duties

Description: The ability to specify that two tasks must be allocated to different resources in a given workflow case.

This pattern is supported by WebWorkFlow. As an example, consider the case in which a paper must be reviewed by someone other than its submitter:

```
procedure pattern(p : Person) {
  process {
    submit(p); review(p)
  }
}
procedure submit(p : Person) {
  who { securityContext.principal != null }
  do {
    p.submittedBy := securityContext.principal;
    p.persist();
  }
}
procedure review(p : Person) {
  who { securityContext.principal != null &&
        securityContext.principal != p.submittedBy }
}
```

8.2.6 Pattern 6 - Case Handling

Description: The ability to allocate the work items within a given workflow case to the same resource.

This pattern is supported by WebWorkFlow. This is done by simply specifying both task actors using the same variable:

```
extend entity Person {
  performingUser -> User
}

procedure pattern(p : Person) {
  do {
    p.performingUser := jack;
    p.persist();
  }
  process {
    step1(p); step2(p)
  }
}
procedure step1(p : Person) {
  who { securityContext.principal == p.performingUser }
}
procedure step2(p : Person) {
  who { securityContext.principal == p.performingUser }
}
```

8.2.7 Pattern 7 - Retain Familiar

Description: Where several resources are available to undertake a work item, the ability to allocate a work item within a given workflow case to the same resource that undertook a preceding work item.

This pattern is supported by WebWorkFlow. This is done by remembering who undertook the previous task, and specifying that only that user can perform the next task:

```
extend entity Person {
  doneFirst -> User
}
procedure pattern(p : Person) {
  process {
    step1(p); step2(p)
  }
}
procedure step1(p : Person) {
  who { securityContext.principal != null }
  do {
    p.doneFirst := securityContext.principal;
    p.persist();
  }
}
procedure step2(p : Person) {
  who { securityContext.principal == p.doneFirst }
}
```

8.2.8 Pattern 8 - Capability-based Allocation

Description: The ability to offer or allocate instances of a task to resources based on specic capabilities that they possess.

This pattern is supported by WebWorkFlow. It can be modeled by including these capabilities in an entity that represents a role or person. For instance, when looking for a valuer with the capabilities to judge the construction of a building, one could use the following model.

```
entity Valuer : User {}
entity ConstructionExpert : Valuer {}

procedure judgeConstruction(b : Building) {
  who { securitycontext.principal != null &&
        securityContext.principal is a Valuer &&
        securityContext.principal is a ConstructionExpert }
}
```

8.2.9 Pattern 9 - History-based Allocation

Description: The ability to offer or allocate work items to resources on the basis of their previous execution history.

This pattern is supported by WebWorkFlow. It can be modeled by constructing a function to assign a task to a user, using an arbitrary algorithm to decide who is the best candidate for executing the task. An example is given in the following model.

```
procedure performTask(t : TaskObject) {
  who { securityContext.principal != null && securityContext.principal == t.performingUser }
}

function assignToBestCandidate(t : TaskObject) {
  for (u : User) {
    ...
```

```
      if (...) {
        t.performingUser := u;
        t.persist();
      }
    }
  }
```

8.2.10 Pattern 10 - Organizational Allocation

Description: The ability to offer or allocate instances of a task to resources based their position within the organisation and their relationship with other resources.

This pattern is supported by WebWorkFlow. Its solution is essentially the same as the solution to pattern 9, except for the particular implementation of the algorithm in assignToBestCandidate which might take organisational position into account rather than an actor's history.

8.2.11 Pattern 11 - Automatic Execution

Description: The ability for an instance of a task to execute without needing to utilise the services of a resource.

This pattern is supported by WebWorkFlow, by defining an automated procedure as follows.

```
auto procedure automatedTask(object : Person) {
  ...
}
```

8.2.12 Pattern 12 - Distribution by Offer - Single Resource

Description: The ability to offer a work item to a selected individual resource.

This pattern is supported by WebWorkFlow. This can be done by allowing a work item to be done by a role (actually a group of people belonging to that role), and notifying only one, by sending an email.

```
procedure doTask(object : Person) {
  do {
    // some algorithm to decide who to notify
    for (u : User) {
      ...
      object.notifyOffer := u;
      object.persist();
    }
  }
  process {
    notifyResource(object)
    ; performActualTask(object)
  }
}

procedure notifyResource(object) {
  do {
    // send an email to user in object.notifyOffer
  }
}
```

8.2.13 Pattern 13 - Distribution by Offer - Multiple Resources

Description: The ability to offer a work item to a group of selected resources.

This pattern is supported by WebWorkFlow. It can be done by using the solution to pattern 12, but instead of choosing one user to notify, notifying all users belonging to a role.

8.2.14 Pattern 14 - Distribution by Allocation - Single Resource

Description: The ability to directly allocate a work item to a specific resource for execution.

This pattern is supported by WebWorkFlow. It is essentially the same as pattern 1: direct allocation if specified at design time. If allocation should be left until runtime, it is the same as pattern 3: deferred allocation.

8.2.15 Pattern 15 - Random Allocation

Description: The ability to offer or allocate work items to suitable resources on a random basis.

This pattern is not supported by WebWorkFlow. The reason for this, is that WebWorkFlow does not have a random generator. If a random generator would be added to WebWorkFlow, it would be easy to model this pattern, as WebWorkFlow has enough flexibility for that.

8.2.16 Pattern 16 - Round Robin Allocation

Description: The ability to allocate a work item to available resources on a cyclic basis.

This pattern is supported by WebWorkFlow by using conditions for the database query when allocating a task to a user:

```
procedure doTask(object : Person) {
  process {
    allocateTask(object)
    ; performTask(object)
  }
}

procedure allocateTask(object : Person) {
  do {
    for(u : User order by u.taskCount DESC, u.id DESC) {
      object.performingUser := u;
    }
    object.persist();
  }
}

procedure performTask(object : Person) {
  who { securityContext.principal != null && securityContext.principal == object.performingUser }
  processed {
    object.performingUser.taskCount := object.performingUser.taskCount + 1;
    object.performingUser.persist();
  }
}
```

In this way, the last user in the for loop will be the one with the least amount of tasks, and the task will be assigned to him.

8.2.17 Pattern 17 - Shortest Queue

Description: The ability to allocate a work item to the resource that has the least number of work items allocated to it

This pattern is supported by WebWorkFlow, by changing the previous example slightly.

```
procedure allocateTask(object : Person) {
  do {
    for(u : User order by u.amountOfTasks DESC, u.id DESC) {
      object.performingUser := u;
    }
    object.performingUser.amountOfTasks := object.performingUser.amountOfTasks + 1;
    object.performingUser.persist();
    object.persist();
  }
}

procedure performTask(object : Person) {
  who { securityContext.principal != null && securityContext.principal == object.performingUser }
  processed {
    object.performingUser.amountOfTasks := object.performingUser.amountOfTasks - 1;
    object.performingUser.persist();
  }
}
```

In this way, the user with the shortest list of tasks will be assigned the task.

8.2.18 Pattern 18 - Early Distribution

Description: The ability to advertise and potentially allocate work items to resources ahead of the moment at which the work item is actually enabled for execution.

This pattern is supported by WebWorkFlow. In fact, it is already done this way in the previously used examples for patterns 12 and 13.

8.2.19 Pattern 19 - Distribution on Enablement

Description: The ability to advertise and allocate work items to resources at the moment they are enabled for execution.

This pattern is supported by WebWorkFlow. It can be modeled by performing the allocation of the work item in the `enabled {}` section.

8.2.20 Pattern 20 - Late Distribution

Description: The ability to advertise and allocate work items to resources after the work item has been enabled.

This pattern is supported by WebWorkFlow. As conditions are defined statically, but can be made to depend on runtime information, and it is possible to change this information from outside the procedure, it is also possible to allocate a work item to resources after they have been enabled.

```
procedure performTask(object : Person) {
  who { securityContext.principal != null &&
  securityContext.principal == object.performingUser }
}

allocateTask(object : Person, user : User) {
  object.performingUser := user;
  object.persist();
}
```

8.3 Data Patterns

As mentioned in section 9.3.3, data patterns focus on four different groups of characteristics: data visibility, data interaction, data transfer and data-based routing. None of these, however, apply very well to WebWorkFlow.

In WebDSL, data is usually visible at any place in the code, provided that there is a reference to the actual object on which the workflow is being performed. However, this reference can always be obtained by simply getting all objects of that entity type from the database. This property of WebDSL makes it easy to access data, but at the same time presents us with a problem if we wish to protect certain data by making it accessible from certain places in the code only. WebWorkFlow does not supply any way of protecting data. The only *scope* in which you can define a variable, so that it is not visible outside this scope, is the scope of a `view` clause and a `do` clause, which are combined in a procedure page. Visibility patterns are thus unsupported.

If a developer should need to use procedure-level variables, the best way of mimicking that behaviour is to extend the entity for which the procedure is defined with a new property in which the data can be kept. This data, however, is just as visible from everywhere in the code, as it is saved in the database.

Data interaction and the corresponding *data transfer* do not apply to WebWorkFlow either. There is no way of passing data from and to workflow elements. The only private data to a procedure is within a generated procedure page. Therefore, the only way of passing data between workflow elements is using the aforementioned entity extension for adding an entity-level variable.

Data-based routing occurs when a workflow's control-flow is dependent on certain data. In WebWorkFlow, technically, all workflow-level data can be altered and as such, influence the control-flow. For instance, by setting a `.processed` variable to false after a procedure has been processed might cause another procedure to cease to be available. This is not a very clean way of implementing workflows, however, as it breaks the normal flow of events. By choosing to bypass the process level, it is possible to use data-based routing. An example of this was given in section 5.4.1: *Connecting Procedures Using Shared Data*.

It might be good to extend WebWorkFlow with syntax for defining procedure-level variables. It would be difficult, however, to access this data in procedures referenced in a process definition, because normally, the connection between these procedures is a one-way connection in the other direction. Moreover, real protection of data in the database from other code areas would demand a severe change in the whole of WebDSL. Therefore, though it would be beneficial to support the different data patterns, it is not feasible at the moment to adapt WebDSL and WebWorkFlow for it.

Chapter 9

Related Work

Workflow is a well explored domain in computer science. As such, there are manifold publications on workflow and workflow management systems. We will discuss related technology here.

First, we will discuss workflow management in general, drawing from an important article by Georgakopoulos et al. [22]. Next, we will introduce the Workflow Management Coalition and will discuss some of their work: the Workflow Reference Model and the process language XPDL. Then we will discuss *workflow patterns*, a set of patterns devised by Van der Aalst et al. [61], to have a way of measuring the capabilities of different workflow management systems. Then we will discuss the workflow description language *BPMN* [72]. We will also discuss *BPEL4WS* [47, 63], an executable workflow language which uses web services to orchestrate a workflow. Next, we will shortly name some formal models for defining workflows, and will discuss Petri-nets in more detail. After this, we discuss *YAWL* [65], a language created by Van der Aalst and Hofstede, focussed on expressing as many control-flow patterns as possible. We will discuss WebWork, a web-based implementation of the Meteor$_2$ workflow management system [42]. Finally, we will look at several languages from which process-aware web applications can be generated: *WebML* [12, 5], *AndroMDA BPM4Struts* [32, 58], and Panta Rhei [15].

9.1 Workflow Management

The subject of workflow management has been well researched by Georgakopoulos et al. in [22].

They distinguish *business processes* from *information processes*. A business process is defined as "Market-centered descriptions of an organization's activities, implemented as information processes and/or material processes". An information process is defined as "Automated tasks and partially automated tasks that create, process, manage and provide information". Once a process is modeled, it can be reengineered. Reengineering a business process is often done to improve customer satisfaction. Information process reengineering, however, is usually aimed at improving efficiency and diminishing cost.

Workflow management is a technology supporting the initial design and reengineering of both business and information processes. It involves defining workflows, and providing for fast (re)design and (re)implementation of the processes as business needs and information systems change [22].

According to Georgakopoulos et al., Workflow Management enforces certain demands on the computing environment of an organization. A *Workflow Management System (WFMS)* should be *component-oriented*, supporting integration of *Heterogeneous, Autonomous and/or Distributed (HAD) legacy and new systems*. It should support workflow applications accessing multiple HAD systems. It should ensure correctness and reliability of applications in the presence of *concurrency and failures*, and support the evolution, replacement, and addition of workflow applications and component systems as processes are *reengineered*.

Many WFMSs however, suffer from the following issues:

- Lack of interoperability among WFMSs
- Lack of support for interoperability among HAD systems
- Inadequate performance
- Lack of support for correctness and reliability
- Weak tool support for analysis, testing and debugging workflows

It is very interesting to notice that these issues are issues that WebWorkFlow also suffers from, and which have to be adressed in the future. A possible conclusion from this fact is that the problems mentioned here are inherent to workflow applications. As not all workflow solutions suffer from these problems, however, we must dismiss this conclusion. BPEL4WS for instance, [1], has great possibilities for interoperability among HAD systems, and workflow solutions based on Petri-nets [60] have good analysis tools for checking correctness and reliability. Some ideas about how to solve these issues in WebWorkFlow are mentioned in section 11.3 on Future Work.

9.2 Workflow Management Coalition (WfMC)

The Workflow Management Coalition (WfMC) is a global organization of parties involved in using and developing workflow and business process modeling. Founded in 1993, they endeavoured to create standards for workflow management system architectures and exchange formats for process definition languages. Results of these endeavours are, amongst others, the *Workflow Reference Model* and the *XML Process Definition Language* (XPDL). We will discuss these in the remainder of this section.

9.2.1 Workflow Reference Model

The Workflow Reference Model was first published in 1995 to provide a reference architecture for Workflow Management Systems (WFMSs), including the most important system interfaces. They intended for these interface specifications to “enable interoperability between heterogeneous workflow products and improved integration of workflow applications with other IT services such as electronic mail and document management, thereby improving the opportunities for the effective use of workflow technology within the IT market, to the benefit of both vendors and users of such technology” [27]. Their approach was to create a general implementation model that matched most workflow products available then, to provide a common basis for developing interoperability scenarios.

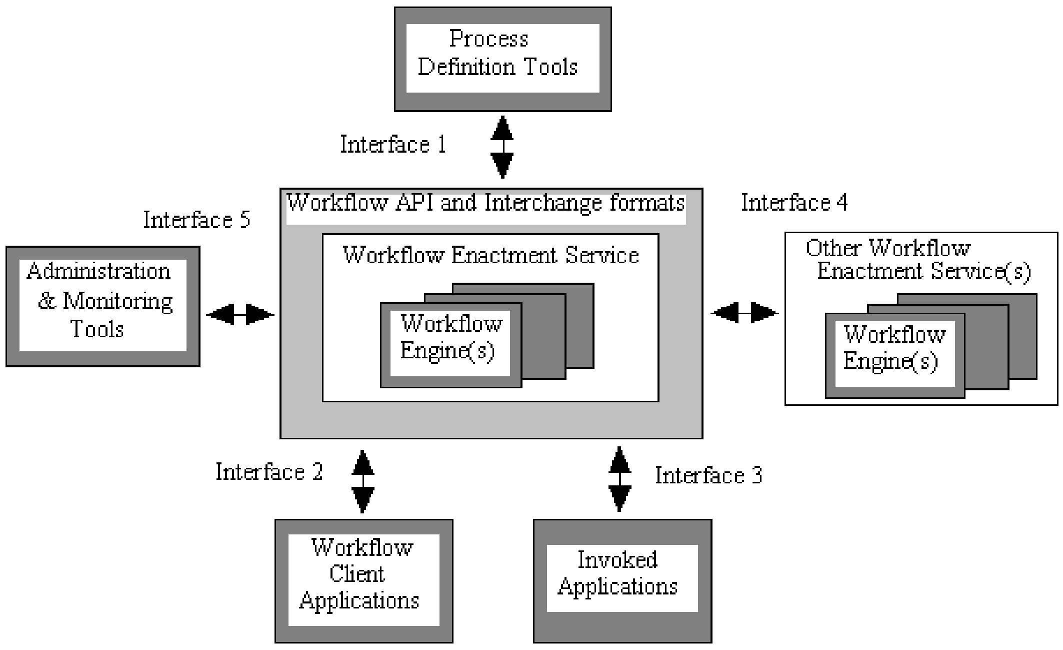

Process Definition Tools
Interface 1
Workflow API and Interchange formats
Interface 5
Interface 4
Administration & Monitoring Tools
Workflow Enactment Service
Workflow Engine(s)
Other Workflow Enactment Service(s)
Workflow Engine(s)
Interface 2
Interface 3
Workflow Client Applications
Invoked Applications

Figure 9.1: Workflow Reference Model

The workflow reference model consists of a *Workflow Enactment Service*, including one or multiple *Workflow Engines*, *Process Definition Tools*, *Workflow Client Applications*, *Administration and Monitoring Tools* and interfaces between the first and the latter three. Moreover, the model defines interfaces to *Invoked Applications* and *Other Workflow Enactment Services*.

A *Workflow Enactment Service* is defined as: “A software service that may consist of one or more workflow engines in order to create, manage and execute workflow instances. Applications may interface to this service via the workflow application programming interface (WAPI)” [27]. It consists of one or multiple *Workflow Engines* (“A software service or ‘engine’ that provides the run time execution environment for a workflow instance.”), that typically handle tasks like interpreting the process definition, controlling process instances, and scheduling deadlines. It adds work items to the user work lists and invokes other applications. The mentioned WAPI represents the 5 interfaces of the workflow reference model, including functions for all 5.

Workflow Client Applications are used by human workflow participants to perform workflow tasks. They consist of a worklist viewer that either presents a work item to a user to be performed, or presents a list of work items from which the user can choose. The Workflow Client Application is also in charge of the user interface that human participants encounter.

Invoked Applications can be used to perform automated tasks in which no user input or interference is required. Typically, this would be done by directly activating a specific tool on the server to execute a task.

Using *Administration and Monitoring Tools*, the execution of workflows can be monitored. By defining a separate interface for this, the WfMC intends for tools to be created for use across different WFMSs.

For specification of workflow processes, often a separate *Process Definition Tool* is created for a WFMS. Using this tool, specification of processes is made easy, and an interface is provided to transfer process definitions to the Workflow Enactment System for execution.

WebWorkFlow does not exactly conform to the Workflow Reference Model. However, we can map the elements of the reference model onto elements of a WebWorkFlow application. Instead of having a designated application for workflow enactment, WebWorkFlow includes workflow-specific logic in its general application logic: there is no separation between workflow control and other application areas. Specification of workflows is also merged into application specification in general. WebWorkFlow, being a web-oriented language, provides adequate facilities for user interface definition. As a Workflow Client Application, one could say a web browser is used. WebWorkFlow does not yet provide any administration or monitoring tools, nor does it give possibilities for integration with other ‘workflow enactment services’. Invoking applications on the server is also unsupported.

9.2.2 XPDL

XPDL, short for XML Process Definition Language, was originally intended to be an interchange format for interface 1 in the Workflow Reference Model [29], between Process Definition Tools and the Workflow Enactment Service. Its new version XPDL 2.0 additionally serves as an interchange format for BPMN. However, as XPDL is expressly not an executable language, it does not fulfill the role of implementation of interface 1.

The WfMC have tried to set the standard and lead the way with XPDL. However, BPMN ended up as being the de facto standard for business process modeling, and BPEL for executable workflow languages, resulting XPDL to be conformed to BPMN.

In [62], Van der Aalst criticizes XPDL for several reasons. One is that XPDL has problems with several of the workflow control-flow patterns [61]: only 11 of the first 20 workflow control-flow patterns are supported. Another reason for criticizing XPDL is that it does not have unambiguous semantics. In the article, seven different ways of joining two parallel paths are discussed, while XPDL only mentions two, without clearly defining their semantics. In [59], Van der Aalst adds that “WfMC’s XPDL is an example of a standard which is imprecise thereby allowing vendors to have their own interpretation of the standard (thus making the standard useless).”

9.3 Workflow Patterns

While there was much on-going development in the field of workflow languages and workflow management systems, there was not yet a universal organizational theory or method for evaluationg the different emerging products. Van der Aalst et al. saw this deficiency and have done a lot of work to create a firm basis for evaluating workflow constructs, based on the idea of *workflow patterns*

[61]. In the absence of a common formalism for workflow modeling, they have taken an empirical approach to identifying common patterns found in different workflow modeling languages [53].

In three consecutive papers, they provided three different sets of workflow patterns for different perspectives of workflow management. The perspectives they cover are the *control-flow perspective* [61], the *data perspective* [53] and the *resource perspective* [52]. Besides these perspectives, the *operational perspective* is mentioned in [61], but no patterns are devised for that perspective. Later, patterns were added for *exception handling* in *process-aware information systems* (PAIS) [56]. For every perspective, a set of workflow patterns was devised. These patterns range from commonly found, basic constructs to more complicated constructs or patterns found in little or none of the workflow management systems available today. Together, they provide a sound basis for evaluation of the workflow management systems and the languages in which the workflow processes are defined.

9.3.1 Control-flow Perspective

The first article, published in 2003 [61], covers the *control-flow perspective*. It describes this perspective as

> activities and their execution ordering through different constructors, which permit flow of execution control, e.g. sequence, choice, parallelism and join synchronization. Activities in elementary form are atomic units of work, and in compound form modularize an execution order of a set of activities.

Van der Aalst et al. provide 20 different patterns for describing a workflow process. They are divided into several categories, namely *basic control flow patterns*, *advanced branching and synchronization patterns*, *structural patterns*, *patterns involving multiple instances*, *state-based patterns* and *cancellation patterns*. After describing these patterns, a comparison between different workflow products is made. 15 workflow products are included in this review. For each of them, support for the stated patterns is evaluated.

Several years after the publication of this first paper, another paper was published containing a revised view on the workflow control-flow patterns [55]. This paper provides a systematic review of the 20 original control-flow patterns, as well as a formal description using Colored Petri Nets [30]. Also, 23 additional control-flow patterns were identified.

9.3.2 Resource Perspective

According to the article on the *resource perspective*, the view on *resources* in *process-aware information systems* has been neglected over the past years [52]. Resources can be either human workers, or non-human resources, such as equipment or computers. Typical factors for the resource perspective include human resources being part of an *organization*, having a specific *position* with associated *privileges*. A resource may be associated with *roles* or have certain *capabilities*.

The resource perspective is much associated with how work is distributed among resources. Two particular terms in this respect are *allocating* a work item and *offering* a work item. When a work item is *allocated* to a specific resource, this resource has a responsibility to finish the work allocated to it. When a work item is *offered* to a resource, or a set of resources, the resource can choose to take upon it the responsibility to perform this work.

The different resource patterns fall into 7 different categories: *creation patterns*, *push patterns*, *pull patterns*, *detour patterns*, *auto-start patterns*, *visibility patterns*, and *multiple resource patterns*.

9.3.3 Data Perspective

The article on *data patterns* identifies a series of patterns that “aim to capture the various ways in which data is represented and utilised in workflows.” [53]. The patterns focus on four different groups of characteristics: *data visibility* (the availability of data elements to various components of the process), *data interaction* (the manner of passing data between elements in a workflow), *data transfer* (the actual transfer of data between workflow components - i.e. by value, by reference etc.) and *data-based routing* (the manner in which data elements can influence control-flow operation).

In chapter 8, we will review WebWorkFlow on the basis of these patterns to evaluate its coverage.

9.4 BPMN

The *Business Process Modeling Notation* (BPMN) [72] is a language designed by the Business Process Modeling Initiative in 2004, and has been maintained by the *Object Management Group* (OMG) [50] since OMG and BPMI merged in 2005. The final adopted specification of BPMN 1.0 was published in 2006 [45]. BPMN is intended to be readily understandable by both business and IT stakeholders. The idea behind it is even to use a single notation for the whole development process: from the documentation of business requirements to the specification and execution of workflows [50]. Altough BPMN itself is not executable, it should be possible to transform a BPMN model into a BPEL4WS model. This is done in [73]. However, according to Wohed et al., this mapping “is only partial, leaving aside models with unstructured topologies as well as constructs such as OR-join and complex gateways” [74].

According to the BPMN Specification Document,

> The primary goal of the BPMN effort was to provide a notation that is readily understandable by all business users, from the business analysts who create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and, finally, to the business people who will manage and monitor those processes. BPMN will also be supported with an internal model that will enable the generation of executable BPEL4WS. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation. [45]

Wohed et al. claim that the BPMN specification is not well formalized, and that in consequence, its semantics are ambiguous, which presents problems when generating executable BPEL4WS [74]. They conclude that BPMN supports a much richer set of control-flow constructs to describe the process. This presents problems when generating a BPEL4WS implementation from a workflow described using BPMN, if the more complicated constructs of BPMN are used, that are not supported by BPEL4WS.

Because of that, generating a BPMN description from a BPEL4WS implementation is not as much a problem as the other way around. Therefore the second goal of the BPMN effort, stated in the following quotation, is more successfully targeted.

> Another goal, but no less important, is to ensure that XML languages designed for the execution of business processes, such as BPEL4WS (Business Process Execution Language for Web Services), can be visualized with a business-oriented notation. [45]

We will now shortly explain and demonstrate the use of BPMN, drawing from [72]. In this article, a more elaborate introduction to BPMN can be found.

BPMN have a similarity to Flowcharts, which will cause business analysts to feel slightly familiar to the language. Just like in Flowcharts, activities are modeled by rectangles and decisions are modeled by diamonds [72]. BPMN has four sets of elements: *flow objects*, *connecting objects*, *swimlanes* and *artifacts*. All different elements are displayed in figure 9.2.

Flow objects are of three kinds: *events*, *activities*, and *gateways*. An *event* signifies that something happens, for instance the start or end of the workflow. An *activity* is an amount of work that is performed. Activities can either be atomic or compound. A *gateway* controls divergence and convergence of sequence flow.

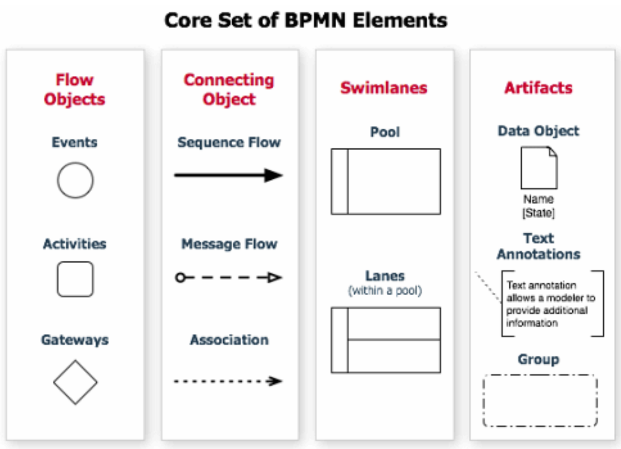


Figure 9.2: BPMN Elements

Connecting objects can represent *sequence flow*, *message flow*, or an *association*. *Sequence flow* depicts the order that activities will be performed. *Message flow* is used to show the flow of messages between two process participants, separated from each other in *pools*. An *association* is used to associate artifacts with flow objects.

Swimlanes are used to organize activities into separate categories to indicate different functional capabilities or responsibilities. There are two types of swimlane constructs in BPMN: *pools* and *lanes*. A *pool* represents a participant in a process. A *lane* is a sub-partition within a pool, extending the entire length of the pool. They can be used for instance for modeling different roles or users within an organization with a pool representing the organization itself.

To add extra possibilities for modelers to illustrate a model, *artifacts* can be used. They do not influence the sequence flow of the process. There are three different types of artifacts. *Data objects* are a mechanism to show how data is required or produced by activities. A *group* is used for documentation or analysis purposes, but leaves sequence flow untouched. Its representation is a dashed line rectangle. *Annotations* can be used to provide extra text information for the reader of a diagram.

Too illustrate the use of BPMN, we have modeled the PDP workflow example described in section 2.5.1. It illustrates the use of swimlanes, flow objects and connecting objects. The organization is modeled using a pool called *company*. It is divided into separate lanes for the roles of *employee* and *manager*. The workflow starts with a split gateway that models the execution of two activities in parallel. After this, the execution flows are joined using a synchronizing join, in which the workflow waits for both branches to finish before continuing. After this, the *Write report* activity is performed. The resulting report is approved or disapproved using the *approve* activity, which will cause the workflow to end on approval, or otherwise cause the report to be rewritten. This choice is modeled by a gateway with two outbound sequence flows, each with a condition regarding the value of *approved*.

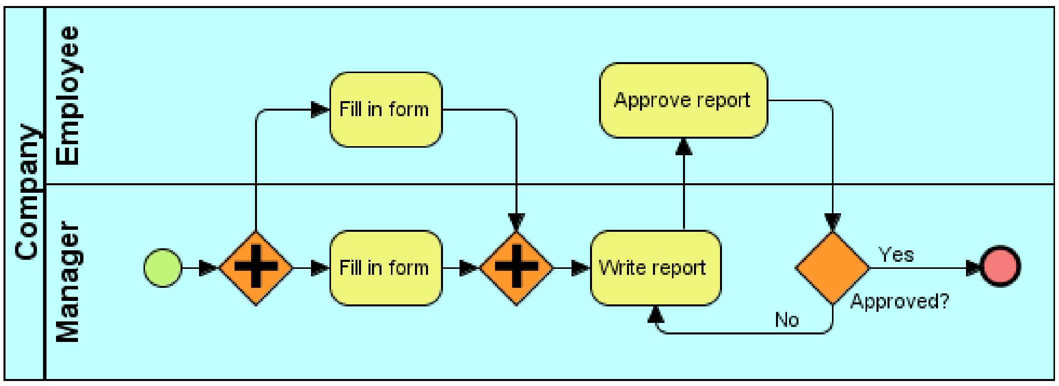


Figure 9.3: BPMN Example

9.4.1 Discussion

BPMN is very expressive in respect to the process perspective, as 16 out of the 20 original workflow process patterns are supported, and one more is partially supported [54]. This is about as expressive as WebWorkFlow.

BPMN has much more constructs available than WebWorkFlow. This contributes towards the clearness of the diagrams, as there are many specific constructs for illustrating the process. It also adds to the elaborateness of designing a BPMN process, because there are more details to attend to. Hoewever, this might cancel out the increased detail that is needed for specifying a workflow for execution.

The fact that BPMN is focused on visualizing workflow processes rather than execution of processes is significant. It results in a good understandability for business analysts, where WebWorkFlow is not comprehensible for non-technical people. Its understandability and expressiveness has made BPMN successful: it is the de facto industry standard for workflow visualizing. Currently, there are 54 implementations of BPMN, and 4 planned implementations. It also has its consequences if any workflow software system is required for IT support for performing the workflow. In that case, another step is necessary: the implementation of a matching software system. By constructing a mapping to BPEL, this process can be largely done automatically, but still semantic ambiguity presents problems that can only be solved by manual intervention.

Different languages have different strengths and weaknesses. That is why the same workflow is best modeled somewhat differently when using different languages. For instance, there are already differences between the WebWorkFlow implementation and the BPMN implementation of a simple workflow like the Personal Development Plan workflow. If we compare our implementation in section 4.1 to the BPMN implementation, we can see that the approvement step is modeled differently. In the WebWorkFlow model, it is easy to model two parallel paths of which only one is executed, using the **+** operator. Therefore, in WebWorkFlow we model the choice between approving and disapproving by presenting two parallel tasks of which one is executed. In BPMN however, it is easier to model it as one activity and to decide what activity to enable afterwards, depending on the result of the previous activity.

9.5 BPEL4WS

The Business Process Execution Language for Web Services, from now on referred to as BPEL, is a language used for describing a business process in such detail that it can be run by a BPEL engine: a workflow enactment system that takes care of scheduling tasks in a BPEL process. Many implementations of BPEL engines exist, among which are IBM BPEL4J, Oracle BPEL Process Manager, and open source implementations ActiveBPEL and Bexee.

BPEL originated from WSFL (Web Services Flow Language) and XLANG, proprietary formats owned by IBM and Microsoft, respectively. It is the de-facto standard for executable workflow languages on the web [63].

Processes in BPEL4WS export and import functionality by using Web Service interfaces exclusively [1]. BPEL was created to provide a way to integrate a diversity of different platforms. Sometimes called a web service orchestration language [48], it is used as a controller to orchestrate different resources scattered all over the web.

BPEL code is specified using an XML format. This XML format is quite verbose and not easy to use. According to Van der Aalst et al. [63], it is a difficult language to use. The many constructs are usable only by experienced users. The graphical representations created using tools made by several vendors, reflect the BPEL code directly, and as such, do not improve on the intuitiveness of the BPEL code to end users.

However, despite these drawbacks, BPEL has managed to become the de facto standard in the industry for executable workflows using web services, inadvertently fulfilling the role that the Workflow Management Coalition has unsuccessfully tried to fulfill for years.

9.5.1 Discussion

Because of the integration-oriented nature of BPEL, specifying a workflow using BPEL will provide you with an executable process model. Being focused on integrating software systems in a workflow, the individual tasks that make up the workflow, have to be implemented elsewhere. If a workflow is to be constructed using BPEL, and not integrating existing systems, this could be seen as drawback, because it is much more tedious to separately develop a control-flow processor and different task runners than to construct a full workflow application as is done with WebWorkFlow.

However, the approach of BPEL also provides significant advantages over WebWorkFlow, as BPEL is able to fulfil the role of an integrating central party for existing workflow, or partial workflow systems, whereas WebWorkFlow, so far, can not be integrated with other approaches. Besides, its focus on the control-flow aspect of workflows might make BPEL a better formed language than it would have been if it included syntax for defining all sorts of workflow tasks.

In our opinion, the reason that BPEL has become the industry standard, is that it is focused on providing means for integrating HAD systems. WebWorkFlow would very much benefit from an interface for use in a service-oriented context. If WebWorkFlow supported web services, it would be able to model existing BPEL processes using WebWorkFlow.

9.6 Formal Process Representations

There have been manifold attempts to capture workflow processes in a formal model. This includes using an existing formal representation and extending it to make it suitable for expressing and executing workflows, as is done in YAWL [65], which we will discuss in section 9.7. In this section we look at formal representations being used for the purpose of doing formal analysis and verification. In [10], an extensive review is done on methods for verifying BPEL process definitions using existing formal analysis and verification tools. Some of these, we will briefly mention here. After this, we will more elaborately look at the use of Petri-nets as a formal representation.

A popular system for software verification is SPIN [28]. Using this tool, models specified using Promela (Process Meta Language) can be verified. In [21], BPEL processes are first translated to an internal representation consisting of guarded automata augmented with unbounded queues for incoming messages. After this, a Promela version of the program can be generated for use with the SPIN Model Checker.

In [20], a tool is developed for generating a VERBUS CFM (Common Formal Model) from a BPEL workflow definition. This internal representation format for processes is based on Finite State Machines. A CFM can be subsequently translated to several formal model formats. These include Promela and SMV [39].

Another way to model a process is using a *process algebra*. According to [10], a process algebra is “a rather small concurrent language that abstracts from many details and focuses on particular features”. Amongst process algebra that have been used to model workflows are CCS (Calculus of Communicating Systems) [43] and its newer brother π-calculus [49].

Abstract State Machines [4] have also been used. This is done in for instance [18].

9.6.1 Petri-nets

One way to formally represent a workflow process, is by using Petri-nets. Petri-nets were invented by the German mathematician and computer scientist Carl Adam Petri. A Petri-net is a directed, weighted, bipartite graph [44] of which the nodes are either *places* or *transitions*. Places are used to model *state*, and transitions are used to model *events*. Arcs are either from places to transitions or vice versa, making it a bipartite graph. The graph has an initial state called *initial marking*, M_0. A *marking*, consisting of a nonnegative integer, is attached to every place and represents the amount of *tokens* in a place.

A transition can *fire*, which amounts to the consumption of tokens from its input places and the production of tokens at its output places. The amount of tokens consumed, as well as the amount of tokens produced are signified by the weight of the connecting arcs. A transition is called enabled if the Petri-net is in a state at which the transition can fire. This is the case if all of the transition's input places contain an amount of tokens that is greater or equal than the weight of the input arcs.

Consider figure 9.4, in which an example Petri-net transition firing is displayed. At the top side of the example, a Petri-net is shown with an enabled transition. When this transition fires, it results in the Petri-net at the bottom side of the example. As clearly visible, two tokens are consumed from the top input place, as well as one token from the bottom input place, corresponding to the arc weights. Two tokens are produced as a result of the transition.

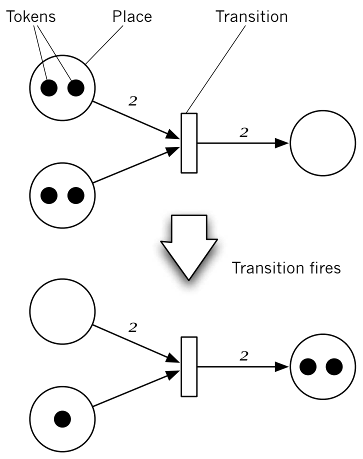


Figure 9.4: Example Petri Net

Petri-nets can be used to model a variety of activities or situations, amongst which are *finite-state machines* and *parallel activities*. For a longer list, the reader is referred to [44]. Parallel activity is easily modeled, due to the possibility for tokens to exist in multiple places at the same time. Presence of a token in a specific place within a workflow graph signifies that an event at an input arc of the place has just happened, and that a transition at an output arc might happen shortly after. An enabled transition corresponds to a workflow task that is available to be performed.

However suitable Petri-nets are for workflow modeling, they still have their limitations when it comes to representing multiple workflow cases. Higher level Petri-nets are necessary for that, and Colored Petri-nets provide a solution [60]. By giving tokens a unique color according to the case they belong to, it is possible to distinguish between different cases throughout the Petri-net during the execution of workflows. In [33], Colored Petri-nets are used to model a workflow process.

In [60], three main reasons are given why using Petri-nets for representing workflow processes is a good idea. In this article, Van der Aalst reflects on a project for the Dutch Customs Department he is involved in, for which a proprietary workflow management system (WFMS) is designed. He presents the reasons for deciding in favour of using Petri-nets as a base of the WFMS. These reasons are *formal semantics despite the graphical nature*, *state-based instead of event-based*, and availability of an *abundance of analysis techniques*. These reasons are further explained here, in the process of which we will compare Petri-nets with WebWorkFlow process and procedure definitions.

1. Petri-nets have a *graphical nature*. This has the advantage that people with no prior experience in computer science are still able to specify workflow procedures [60]. Despite the fact that they are graphical in nature, Petri-nets still have *formal semantics*. This has a multitude of advantages. For one, the procedure specifications are unambiguous. As such, they can serve as a contract between subdepartments, and can be used to resolve conflicts that may arise. The representation benefits from a tool-independant interpretation. Moreover, it allows for reasoning about properties like the absence of deadlock, and makes it possible to apply analysis to the workflow.

   WebWorkFlow, being a textual language, is fundamentally different on this point. A disadvantage of WebWorkFlow not being a graphical language is that non-technical people with not easily be able to understand a WebWorkFlow model. On the other hand, despite other advantages, the textual representation is decidedly formal, which gives it all the advantages of Petri-nets just mentioned.

2. Petri-nets are state-based. Many workflow systems use representations that are event-based. In event-based representations, however, it is very difficult to model state. As only events are explicitly signaled, it follows that the state is defined implicitly. According to Van der Aalst, state-based representation have several advantages over event-based representations:

   - It allows for a clear distinction between *enabling* and *executing* a task. Only automatic tasks are immediately executed at the moment they are enabled. Other tasks are dependent on human intervention, and have separate moments on which they are enabled and executed. This is clearly distinguishable in state-based modeling of workflows.
   - Separate enabling and execution of tasks creates the possibility for *competitive tasks*, in which only one of two tasks may be executed despite the fact that both of them are enabled. In state-based modeling of workflows, it is straightforward to revoke the enabling of the task that is not executed in the end.
   - Workflow cases may need to be moved from one local system to another, during execution. When using a state-based model, this is easy to do. When using Petri-nets, all that needs to be done is transferring the tokens to the equivalent system on another location. Exchanging cases between event-based WFMSs is more difficult.

   WebWorkFlow is neither state-based or event-based. In its process language, procedures are scheduled in a control-flow description that decides the order of events. WebWorkFlow clearly distinguishes enabling and performing a procedure. This notion is also captured in the process abstractions. For instance, the deferred choice construct enables both branches, but only one of the branches will be executed in the end. Thus, it also supports competitive tasks. Moving workflow cases from one system to another could probably be organized by moving the entity object on which the workflow is running to the other system, and also moving any workflow status object referring to that object.

3. Petri-nets have been around for a long time, so many tools have been developed to perform analysis on Petri-nets. Instead of developing custom analysis techniques for a workflow modeling language, it is possible to use previously developed, widely used and tested, analysis tools. Examples of the use of analysis for workflows are manifold. Analysis can be done to *prove properties* of the workflow, like invariance properties, or deadlock. Using analysis, we can calculate *performance measures* like response times, waiting times and occupation rates.

   Analysis techniques used by Van der Aalst for the Dutch Customs Department project include proving in polynomial time that a workflow is a *sound* workflow procedure. Soundness is defined as fulfilling the following requirements:

   - There are no tasks that do not contribute to the processing of cases.
   - For any case, the procedure will terminate eventually.
   - At the moment a case terminates, all references to it have been removed.

   Also, by simulating a workflow, it is possible to evaluate the workflow performance without actually enacting the workflow.

   WebWorkFlow presently provides no possibilities for applying analysis to processes or procedures. In this respect, using Petri-nets is favourable to using WebWorkFlow. This advantage of Petri-nets over WebWorkFlow is due to its being a well-established technology.

Several attempts have been made to use formal modeling languages for modeling BPEL processes. In [64], Van der Aalst et al. demonstrate the use of Colored Petri-nets for this objective. In the article, which is in fact a case study, they describe the whole process of starting with requirements and ending up with a working BPEL implementation of a workflow system. They start by creating a Requirements Colored Petri Net (RCPN) to model the situation. This subsequently transformed to make it suitable for conversion to BPEL. Doing this, they make sure that the restricted version of the RCPN, called Colored Workflow Net (CWN), adheres to the following requirements:

- The workflow model should use a *restricted vocabulary* common for workflow management systems. Only concepts such as case, task, resource, role, organizational unit, and attribute may be used to contruct workflow models [64].

- The workflow model includes only actions that are to be supported by the workflow management system. Any manual actions the system will not be aware of, will be stripped from it.

- The workflow model refines selected parts of the requirements model to enable system support. This may include improving the level of detail compared to the requirements level.

After transforming their RCPN to a CWN, they generate BPEL code templates for implementing the workflow. These templates need only interface definitions to connect to existing applications and data structures.

While this process is well-described and contains a formal modeling of the requirements, which might also be useful for other applications, it is tedious work to develop two separate Petri-nets during the process.

It is not possible to do complete code generation. This has both advantages and disadvantages. Partial code generation can be useful in some situations, to add implementation-level specifics to the product of a model that is in this way kept clean. A model from which a complete application is generated needs to be more detailed and specific than a model from which a partial application is generated. On the other hand, partial code generation presents problems if the original model is changed: the changes to the generated code might be lost.

9.7 YAWL

When reviewing 15 different workflow management systems using the control-flow patterns described in [61], Van der Aalst et al. noted that no workflow management system supports all patterns. This triggered them to see if they could do a better job. Starting with Coloured Petri-Nets [30], they devised a new language called YAWL [65], that supports all 20 original control-flow patterns except one. While Coloured Petri-Nets already support more patterns (namely, 14) than most workflow management systems, YAWL improves on that by first identifying the primary problems with Coloured Petri-Nets, and subsequently introducing new constructs for Coloured Petri-Nets to battle these problems. The three areas in which the problems fall are *patterns involving multiple instances*, *advanced synchronization patterns* and *cancellation patterns*.

YAWL includes a system to specify users and roles for use in the workflow specification.

9.7.1 Discussion

YAWL is an enormous improvement on for instance BPEL, in respect to the control-flow patterns supported. It supports 19 of the first 20 control-flow patterns, which is a very high score. This is not surprising, considering that YAWL was designed by the workflow pattern creators. Despite supporting many workflow patterns, the language only has relatively few different constructs. This leads us to conclude that the available constructs are very expressive.

Another great benefit of YAWL is that it is based on a formal process representation: Colored Petri-nets. This allows the user to do all sorts of analysis on workflow definitions.

YAWL has a service-oriented architecture, which means that it can be integrated in other, existing workflow systems. This is a significant advantage over WebWorkFlow.

However, YAWL still has some disadvantages. For one, the specification of a workflow using YAWL uses a graphical tool in which many properties of the different activities have to be specified using hidden property screens in which XML is used for data definition. This leads to a view on the whole process in which the details are not clear. Also, workflow specification becomes an elaborate process.

Another disadvantage of YAWL is the limited possibilities for defining a suitable user interface. It has a web interface that supports automatically generated forms. A running workflow is advanced by entering information in forms specified using XML, but the interface is not very intuitive.

9.8 Meteor$_2$ WebWork

WebWork, developed by Miller et al., is a web-only implementation of the workflow management system Meteor$_2$ [42]. The other implementations are OrbWork and NeoWork. Both are implemented using CORBA, and they differ in the fact that OrbWork is a distributed version of the system, whereas NeoWork is a centralized version. WebWork is a slightly downscaled, light-weight version.

WebWork has a separate designer tool called the METEOR Designer (MTD). It is a graphical tool consisting of a map designer, a data designer and a task designer.

Miller et al. look upon a workflow enactment system as providing the command, communication and control for individual *tasks* participating in a workflow. They see tasks as run-time instances of intra- or inter-enterprise applications. Tasks are executed in the following way. A task is started, with some input data, by starting its task manager, a CGI program that coordinates the task execution. The task manager executes an executable program that performs the actual task. After execution, a web page will be viewed; which page exactly depends on the success or failure of the task execution.

Tasks are scheduled using inter-task dependancies. An intertask dependency consists of a *dependency arc*, which signifies the next task to be enabled, a *dependency condition*, to specify conditions for enabling the next task, and possibly a precondition for joining when multiple input arcs are present. This is a boolean expression of ORs and ANDs. Using ORs, any input arc can enable the task. Using ANDs, all input arcs must give the enable signal before the task is enabled.

9.8.1 Discussion

On some points, WebWork has an advantage over WebWorkFlow. The most promintent advantage lies in the fact that its architecture is open and hence, it is able to communicate with other systems and to form a workflow system containing other HAD systems. By allowing the system to execute any executable as part of a task really gives wide opportunities for integration.

The advantage of WebWorkFlow lies in the fact that due to its integrated workflow definition approach, it is possible to specify workflow and task implementations in the same language, providing a straightforward way to create a fully working system in one specification.

9.9 WebML

WebML is a language for designing web applications using a graphical editor. It provides separate definition of a *data model* and of *site views*, expressing the web interface that is used to view and manipulate the data. From a WebML application model, a fully functioning web application is generated. More on WebML can be found in [12].

Being about data-centric web applications, WebML is a system that compares to WebDSL well. Brambilla et al. [5] have extended WebML with constructs for workflow. They mention that it is possible to make a process-oriented application, in which the process-oriented parts and the data-centric parts are integrated, an approach they call *implicit process control*. They distinguish two main ways to do this. The first is by letting pages correspond to tasks, while the process description is expressed using hyperlinks between the pages. These hyperlinks are viewed conditionally, so that users can only perform their own tasks. Splits are constructed using multiple conditional hyperlinks to different pages, and joins are constructed by linking multiple pages to the same page. Iteration is also possible using conditional hyperlinks. AND-splits, however, are not possible as a user can only follow a single path through the workflow. Also, the workflow can involve one user only. The second way to model implicit process control is to encode the process state into data that is shared between users. This will enable the construction of workflows that involve multiple users. The idea is to encode case advancement, so that hyperlinks to pages on which tasks can be performed, can be displayed under certain circumstances only.

However, mingling process-oriented data with a data-centric application has severe disadvantages. As the data and constructs are not separated, any requests for changes in the process-oriented part of the application will force the developer to find out which elements of the data model belong to the process-oriented part, before it can be changed. To remedy this, they present a process reference model in which process metadata and user metadata are added to the application's native data model. They distinguish processes and activity types from cases and activity instances, respectively, the latter two being instances of the former two. WebML is further extended with primitives for inserting process elements into the site views.

As many workflow applications operate in the context of heterogeneous systems, communicating using web services, it is good to have support for web services in a workflow system. In WebML this is realised by adding constructs corresponding to the primitives offered by WSDL [13]. This makes it possible for WebML to function as a process controller collaborating with other web service-enabled systems, or as part of a distributed process involving multiple process controllers.

In the article, a workflow implementation is derived from a workflow specification in BPMN [72]. This is done by manually creating a WebML application according to the process prescribed by the BPMN specification.

9.9.1 Benefits

WebML has many advantages over regular design of web applications:

- It generates a full-blown application from a model. WebML uses intuitive constructs using which modeling a complex web application can still be comprehensible.
- Separating process entities from the rest of the data model yields a model in which there is no cluttering of process data with the domain data. Also, any changes to the process are much easier.
- WebML gives the possibility to develop integrated workflow applications, such that only one model is used for generating the whole application. This is an advantage over process engines that can only direct the flow of control, while unable to perform the actual tasks, or present a user interface for human participants to perform tasks.

- WebML is quite expressive, as all BPMN constructs can be modeled using WebML constructs.
- WebML has support for web services, which enables it for use in the context of a heterogeneous workflow system.

9.9.2 Disadvantages

There are some disadvantages to the use of WebML for modeling process-oriented applications.

- Access control is part of the extension for workflow, which means that any other use of access control must fit with the workflow access control model, or otherwise two access control systems will have to be used separately.
- The translation from BPMN to WebML workflow is a manual step, which means that the WebML workflow has to be built from scratch. They do mention their vision to create the possibility to derive a skeleton application from a BPMN workflow one day. Unfortunately, deriving skeleton applications that are altered before they can actually be used present problems in case of future workflow changes. Using scaffolding, subsequent regeneration of the model will undo the modifications to the previous version.
- In the current implementation of WebML, process-enabled applications are created as a *process-aware* application, so that the necessary process data models are automatically generated at the start. However, if these models change due to improvements of WebML, any existing WebML Models have to be manually updated, as the process data models have already been generated.

9.10 AndroMDA BPM4Struts

The AndroMDA system (pronounced as Andromeda) is a software generation system using a model driven architecture [32]. For AndroMDA, UML (Unified Modeling Language) diagrams are used to generate code for several different platforms. For each platform to output code for, AndroMDA *cartridges* must be provided. These cartridges can be developed and inserted into the system.

The BPM4Struts cartridge [58], which is short for Business Process Modeling for Struts, is a cartridge that gives possibilities for deriving application pages from process models. It uses UML *activity diagrams* in the source model to describe the process. The are used to specify the behaviour of *use cases*. Added to this is a class diagram that specifies the classes that comprise the model. In the class diagram, there are three tiers: an entity tier, of classes that are persisted using a database, a service tier, which makes up the business-logic, and a controller layer, in which the controllers for the different use cases are specified. AndroMDA uses output in the XML Metadata Interchange (XMI) file format of almost any modeling tool [32]. It generates source code for a web application, including a user interface derived from the stereotypes specified when modeling the use cases in UML. For instance, there is a stereotype for adding a use case to the first page of the application, and another stereotype for specifying that a use case must be added to the user interface navigation section.

After using AndroMDA to generate source code, a small amount of manual coding needs still to be done: the implementation of the activities specified.

9.10.1 Benefits

AndroMDA has some advantages over other approaches:

- As UML is a language that is extensively used within the software engineering community, the choice for UML activity diagrams for describing a process is advantageous.
- The open architecture of AndroMDA provides for good extensibility.
- Because the generated code is not finished yet, it can be manually extended. This guarantees one will never have the problem of not being able to implement a certain feature for a system that is developed using AndroMDA, as all customizations can be added later.
- The user interface is generated from the UML diagrams. This means the developer does not have to worry about building a user interface.

9.10.2 Disadvantages

There are several drawbacks to the AndroMDA approach as well:

- Partial generation forces one to implement part of the application later. As such, the developer has to be careful to provide for a clear separation between generated code and the customizations developed later. Otherwise, it is possible that the customizations are lost when the code is regenerated.
- Partial generation is also a drawback because the developer will have to use multiple languages during implementation, and the application cannot be constructed using only a model.
- The fact that the user interface is inferred from the UML models brings with it that it is not possible to use a fully customized user interface.

9.11 Panta Rhei

The workflow system Panta Rhei [15] is a web-based workflow system that does dynamic interpretation of workflow process definitions specified in the textual language called WDL (Workflow Definition Language). The workflow system handles user interaction using a web front-end. Web pages are automatically generated for forms.

In Panta Rhei, all concepts are first class citizens: agents, forms, activities and even process definitions. This has several advantages, of which one is that the actor for performing a task can be set to a value filled in on the form belonging to the previous task, which makes the execution very flexible.

All process information is mapped to the data space, which means that the process can be manipulated by changing the data. Because processes are interpreted, it is possible to even store processes in the database, and change them during execution.

Assignment of agents to activities can be done in various ways. It can be done by specifying a specific user, by specifying a role - causing any agent fulfilling that role to be able to perform the activity, or by, as mentioned before setting the agent to a value on a form that is filled in during run-time.

Panta Rhei also has a transaction support system. It allows the developer to specify a compensation scheme for rollback, that is started in case a specific activity fails (a developer can also specify conditions for an activity to succeed). Different types of compensation are distinguished. Type *none* means nothing has to be compensated to perform a rollback. Type *undoable* means that a rollback can be performed without any side-effects, for instance by performing an inverse operation. Type *compensatable* means that an action can be performed so that the situation is back to normal again, for instance by transferring back the money that was transferred during the aborted activity. Finally, type *critical* means that the activity cannot be compensated.

9.11.1 Discussion

Panta Rhei has an important advantage over WebWorkFlow: *Dynamic process interpretation* means that workflow process definitions can be changed, not only between execution of different workflow instances, but even during execution of a workflow. This is a significant benefit over WebWorkFlow, as WebWorkFlow exhibits problems when changing the process: the data model will change, as well. Another benefit over WebWorkFlow is the notion of time: WebWorkFlow is not yet equipped with timed processes.

Many other benefits of Panta Rhei are also present in WebWorkFlow. Just like in Panta Rhei, it is possible to assign activities to roles. Also, agents can be selected during runtime by basing a procedure's actor on information that is changed somewhere earlier in the process.

Transactions are something that WebWorkFlow would really benefit from. However, the transactions implemented in Panta Rhei have to be manually constructed by the developer, as opposed to rollbacks being automatically taken care of during runtime. Therefore, we consider this no serious benefit over WebWorkFlow.

Panta Rhei also has some disadvantages compared to WebWorkFlow. For one, conditions for availability of procedures are even more flexible in WebWorkFlow than agent assignment is in Panta Rhei. Another significant deficit of Panta Rhei is the impossibility to define a custom user interface. Being a workflow-enabled web application generation system, WebWorkFlow has enormous flexibility in user interface definition.

Chapter 10

Evaluation

In this chapter we evaluate the results and applicability of WebWorkFlow. We do this by first comparing WebWorkFlow with other workflow languages. Because WebWorkFlow is not only a workflow language but also a domain-specific language, we evaluate the language from that perspective, as well. Finally, we will try to place WebWorkFlow's abstractions in a broader context, by thinking about what elements of the language could be used in other situations than as an extension to WebDSL.

10.1 Comparison with other Workflow Languages

To form an idea of WebWorkFlow compared to other workflow languages, we look at some characteristics of WebWorkFlow and compare these to properties of other workflow languages. We will look at executability, integration, whether it is textual or visual, architecture, application generation and the availability of timed events.

10.1.1 Executable

Contrary to some workflow languages, WebWorkFlow is focused on describing executable workflows. A WebWorkFlow model entails enough information to derive a complete application for executing the workflow. This ensures that no manual effort is needed to convert a non-executable workflow model into a model using an executable workflow language, as is done in [73]. As such, the problem of having to translate ambiguous control-flow constructions into non-ambiguous ones, is evaded.

Because its focus on executability, WebWorkFlow has sufficient detail for execution. This also means that it has more information than generally necessary for a clear business view of the process and as such, is less useful in a context where non-developers are involved. In other words, it might be difficult for non-technical people to understand the workflow by looking at its WebWorkFlow implementation.

10.1.2 Integrated Approach

Some workflow languages only provide constructions for describing the control-flow perspective of a workflow. This is done in *BPEL4WS*, as well as in *BPMN*. In WebWorkFlow, however, a workflow description does not just entail a control-flow description of the process, but all tasks that make up the workflow are also included. This approach ensures a straightforward definition of a complete workflow application.

10.1.3 Textual Language

WebWorkFlow is a textual language, as many other programming languages are. Most developers are to expected to be familiar with using textual language for developing applications. However, most managers and business analysts, being the domain experts involved in the process of requirements engineering and in checking the developed workflow, are used to assessing diagrams instead of code blocks. Explanation of the WebWorkFlow specification will be necessary for business analysts and managers to comprehend the workings of the workflow application.

This is a disadvantage of a textual language for workflow description. A benefit, however, is that having a textual language will make sure that the workflow application will actually be developed by developers that know about software engineering as opposed to people that know business, but might underestimate the intricacies of engineering software.

Another advantage of a textual language is that it is not necessary to develop a software tool to specify the workflow; enough text editors already exist. This means it is not necessary either, to devise a representational format, or design a conversion to XML or another intermediate format. Moreover, version control systems handle textual specifications very well, while graphical notations might present problems for version control systems.

10.1.4 Multiple Abstraction Layers

As frequently mentioned, most workflow languages only have a process-oriented view of the workflow. WebWorkFlow, however, allows for specifying *procedures*, as well as *process descriptions*. Having these two separate layers of abstraction, WebWorkFlow is much more flexible when it comes to custom patterns that are somewhat out of the ordinary. It allows for adding more detail to the workflow definition than could be done if we could only use a process description. As a result of this layered architecture, the process language is expressive, but concise, and the procedure level adds flexibility and even more power.

10.1.5 Generation of Workflow-Enabled Applications

Many workflow systems fall in the category of Workflow Management Systems (*WFMSs*). A WFMS is an application that accepts workflow definitions for execution, keeping track of events and input and ensuring a valid execution of the workflow. Often, WFMSs also include means to define workflows using built-in tools [22]. WebWorkFlow uses another approach: instead of having a separate

WFMS and a workflow definition, a workflow definition is *compiled*, resulting in a standalone workflow application with incorporated business logic.

Although it is very simple, this approach has several disadvantages:

- It is not possible to alter a workflow definition and subsequently feed this new definition into a WFMS that is already running. Instead, a change in the workflow definition will force the developer to abort execution, recompile the workflow application and deploy the new resulting application.
- When changing a workflow definition, its data model is likely to change, as well. When deploying the recompiled application, this will the result in the need to redeploy the database as well, causing the loss of the data so far. Currently, there is no way to automatically migrate the database to the new situation while preserving the old data. This problem is not merely a WebWorkFlow problem, but a problem exhibited by most DSLs that are subject to evolution. A solution to this problem is suggested by Vermolen: heterogeneous coupled evolution, see [67].
- Normally, a WFMS does not just execute workflows, but also provides means for testing, analysis, and monitoring of workflows, as is mentioned in section 9.6. Some WFMSs also provide concurrency control, recovery and transaction coordination [22]. WebWorkFlow does not provide any of these services.

10.1.6 Timed Activities

Many workflows support the notion of timed activities. In many situations, it is beneficial to be able to use timed events to trigger workflow tasks. For instance, a workflow triggered by an employee's birthday might come in handy for companies that wish to buy their employees a present for their birthday. Another example of the use of timed activities is when a certain stage of the workflow should last for only a certain time, for instance two weeks, before the workflow is to automatically advance to the next stage, without human intervention.

So far, WebWorkFlow does not support timed activities, due to the absence of any scheduled activity or action in WebDSL, the underlying technology for WebWorkFlow.

10.2 DSL Evaluation

WebWorkFlow not only is a workflow language; it is also a domain-specific language, to which the evaluation criteria of DSLs apply. In this section, we will first look at a number of criteria that can be used for assessing the quality of DSLs, drawing from [69]. Next, we will apply these criteria to WebWorkFlow and draw our conclusions.

10.2.1 Criteria for DSL Evaluation

Domain-Specific Languages can be evaluated using the following criteria: *expressivity*, *coverage*, *completeness*, *portability*, *code quality*, and *maintainability* [69].

Expressivity relates to the result of the effort in terms of lines of code saved. When a DSL is used, the code size should be decreased compared to using a conventional, general-purpose programming language. This reduction of programming effort is caused by the generation of boilerplate code.

Coverage is the measure in which the language succeeds to capture the domain using the abstractions it provides. A DSL with high coverage will allow a developer to build any type of application for the domain indended.

Completeness relates to the result of the generation. Some code generators generate incomplete code, an approach that is sometimes called *scaffolding*. This metaphor is derived from building construction, in which a temporal wooden construction is used as a skeleton for a building. This skeleton is discarded later, when a firmer and more durable construction has replaced the original one. Generating incomplete code makes building a compiler easier, as any difficult transformations can omitted so the code section concerned can be left for the developer to fill in later. However, this has severe disadvantages for the process. In effect, this approach uses *one-time-only generation*, and adapting the generated code makes it impossible to adapt the original model and regenerate, as the changes to the original generated code will be lost. Introducing protected regions in the source model, so custom code can be included in the model, is a solution for this problem. This however, has other disadvantages, as the implementation of the compiler is exposed to the developer. Apart from this, the developer has to develop using two different languages, whereas using complete code generation, the developer will only have to think about developing using the DSL.

Portability is another measure for evaluating a DSL. If the language is too dependant on the underlying technology, portability is diminished. Avoiding *leaky abstractions*, in which dependencies on the target environment are incorporated in the DSL, is a key issue. An approach to improve portability is the use of a core language, which is explained in section 2.7.4. This will make generating code for different platforms easier.

Code quality of the generated code is important, too. Despite the fact that generated code should in principle not need to be read by humans, it is good to generate well-organized source code, for the purpose of making compiler development easier. Another key issue in which code quality is important, is the efficiency of generated applications. Inefficient code will prevent people from adopting the DSL for use in developing industry-quality applications.

Maintainability of the source model is another metric used to evaluate a DSL. It is good when the design of a DSL enables a developer to maintain a source model in a straightforward way, instead of having to alter much code to implement a single change.

10.2.2 WebWorkFlow Evaluation

In this section, we will shortly apply all aforementioned criteria to WebWorkFlow, and draw a conclusion of the results.

Expressivity: The relatively simple process language provided by WebWorkFlow is very expressive, as most frequently used patterns are directly represented by straightforward abstractions. While defining behaviour on the procedure level is slightly more cumbersome, it is still a relatively concise way of specifying control-flow.

Coverage: As seen in chapter 8, WebWorkFlow's data pattern coverage is very low due to WebDSL's characteristics and the absence of ways to protect data. Coverage of the other perspectives is much higher, though. Most of the *control-flow patterns* are covered. Only two patterns cannot be modeled in WebWorkFlow. Some of the control-flow patterns, however, require a workaround, mostly to handle multiple instances of a workflow or task. The *resource patterns* are covered even better. Only one pattern is not directly supported, because WebDSL does not have a random number generator. All other resource patterns are particularly well modeled using WebWorkFlow.

Completeness: WebWorkFlow draws on WebDSL's compiling method, which uses complete code generation. This is favourable to other types of code generation.

Portability: WebWorkFlow, using WebDSL as an intermediate target language, benefits from WebDSL's core language, which makes for high portability. Adding another target technology can be done by only providing a transformation from the relatively simple core language to the target platfor. The fact that several different target platforms have already been used for WebDSL makes clear that *leaky abstractions* are not a problem with WebDSL.

Code quality: Not much attention has been directed to the optimization of generated code. Therefore, the generated applications are very large. Moreover, a relatively simple application can still be quite slow. These two facts show that WebWorkFlow's generated code quality still needs improvement, as it is not yet up to industrial standards.

Maintainability: Because of Stratego/XT, the technology used for building the WebWorkFlow compiler, the language is easy to maintain. Stratego/XT allows for both horizontal and vertical modularity (see section 2.7.3 for more details), which means that it is possible to extend the compiler while hardly touching the current code. However, there are problems with the data model when changing the language. This is the same problem already mentioned in section 10.1.5.

Except for the code quality, the evaluation of WebWorkFlow using these criteria is overwhelmingly positive. We will draw the conclusion that, from a DSL perspective, WebWorkFlow is a good language, and would profit from improving WebDSL's generated code quality.

10.3 Process-Oriented Abstractions in a Broader Context

WebWorkFlow draws heavily on the use of WebDSL as an underlying key technology. In this section, we will shortly focus on WebWorkFlow as a sub language, and make some remarks on the usefulness of WebWorkFlow's abstractions for different language platforms.

10.3.1 Reusable Aspects

We are convinced that some elements of WebWorkFlow do not depend on WebDSL, and can be successfully reused in extending other languages with workflow-enabling abstractions.

- The advantage of having two different layers of abstraction can most probably be reused for other languages. It really is an advantage to have a simple but expressive process language combined with more detailed, flexible procedure level customizations. Suitability for a separate procedure level is not confined to WebDSL, as it merely provides atomic procedures on objects.

- Using a textual representation for process descriptions is reusable, as well. As discussed, using a textual language has both advantages and disadvantages. However, the simple textual process language in WebWorkFlow is perfectly usable in other contexts, optionally adapting it for a new host language's style.

10.3.2 Aspects More Difficult to Reuse

Some elements of WebWorkFlow are not as easy to reuse in other language, or depend heavily on the characteristics of the specific language that is used.

- The integrated approach that makes WebWorkFlow a language in which it is possible to specify a complete workflow application including the actual tasks of which the workflow consists, is difficult to maintain for some languages. WebDSL is very suitable, as it is a fully-fledged programming language providing all necessary elements for defining workflow tasks. Not all languages to extend with workflow abstractions can be expected to be as suitable as WebDSL.
- The fact that WebDSL is a language for web applications including a custom user interface gives WebWorkFlow a lot of flexibility in this respect. It might be more difficult to design a custom user interface in other languages.
- The suitability of WebWorkFlow for capturing resource patterns is largely due to its strong access control language. When using the abstractions of WebWorkFlow for other languages than WebDSL, it might turn out that specifying actors and other conditions is much more difficult than is the case with WebWorkFlow.

Chapter 11

Conclusions and Future Work

In this chapter we begin by clearly stating the contributions made by this work. After this, we draw conclusions from our work, reflecting on the results of designing WebWorkFlow and performing a case study. We compare the results to the goals we set in chapter 1. We finish by discussing several ideas for future work.

11.1 Contributions

In this section, we briefly state the contributions of this work.

1. We present WebWorkFlow, a textual workflow language based on WebDSL, consisting of two different layers of abstraction. It has a *procedure layer* to define atomic operations involving a page definition, actors, conditions for availability and an associated action. This is extended with a *process layer* to combine previously defined procedures in a *control-flow* definition for the process. This approach is *integrated* and results in complete code generation, allowing for complete definition of a workflow application including actual task definitions, leading to a fully working application.

2. We contribute a method to add process and procedure abstractions to a language to make it workflow-enabled.

3. We provide a case study in which WebWorkFlow, as well as WebDSL, is applied to a real-life situation to evaluate the language.

4. We contribute `string-select-entity`, a minor improvement for WebDSL that results in much more concise syntax, useful for certain kinds of data-intensive web applications.

11.2 Conclusions

In chapter 1, we stated the following requisite aspects for the language:

- Process-oriented abstractions for web applications, involving the notions of *tasks*, *participants* and a *process description*. The majority of regularly used control-flow patterns must be covered in the process description.

- Complete generation of applications, in which both the process description, the implementation of individual tasks and a custom user interface can be specified using a single language.

- Flexibility in respect to the scheduling of workflow tasks. This involves having multiple abstraction layers with the process specification at the highest level, while making available the constructs of lower abstraction layers within higher layers.

The process layer provides the necessary abstractions for process-oriented websites, including the notions of tasks, participants and process descriptions. By performing the coverage analysis in chapter 8, we have established that the language at least covers the majority of regularly used control-flow patterns.

By virtue of using WebDSL as a foundation to build upon, complete generation of applications is made possible, in which the process description, task implementation and user interfaces are specified using a single language: WebDSL augmented with WebWorkFlow constructs.

Flexibility is introduced by having both a process layer and a procedure layer. The process layer, while being the higher-level abstraction, can still use constructs from lower abstraction layers. Therefore, control-flow specification is not limited to the constructs offered by the process layer.

This thesis investigated the following three questions: 1) What is a good design for a language that fulfills the requirements stated above; 2) How to implement such a language; and 3) How does this language work in practice? The first two questions have been answered by developing WebWorkFlow. The third question is answered by performing the LIXI valuation workflow case study.

In the remainder of this section, we will recollect some of the advantages and drawbacks of WebWorkFlow as a language for workflow-enabled web applications.

11.2.1 Advantages

- The approach to provide two different layers of abstraction to form the workflow extension to WebDSL turns out to be favourable. It leads to a process language that is both concise and expressive. Because of the availability of a procedure level, any patterns that cannot be expressed solely using process expressions, can still be expressed using procedure-level amendments.

- Having a language to build on is a big advantage. By allowing the workflow developer to use WebDSL all throughout the workflow definition, it is easy and natural to define a fully working workflow application including individual tasks, while many workflow languages that are firstly focused on the control-flow perspective, have a separate way of defining the individual tasks.

  Another advantage of having a language to build on is that many issues with the language can be treated as orthogonal to the workflow language and as such can be solved separately, without influencing the workflow language design. Most of the future work mentioned in section 11.3 concerns issues orthogonal to the workflow language.

- The approach we used for extending WebDSL with abstractions for workflow can be used for other languages as well. Its success however, will depend on the language in question, as not all languages may prove to be as suitable a language as WebDSL with its action code, its possibilities for defining a custom user interface, and its strong access control language.
- The LIXI valuation workflow case study has provided us with valuable insights and experience with using WebWorkFlow in real-life contexts. On the whole, WebWorkFlow turned out to be suitable for use. Also, a substantial improvement in expressiveness has been achieved, as our implementation of the valuation workflow uses 5 to 6 times less code than the original implementation.

11.2.2 Drawbacks

- Different from many workflow solutions, WebWorkFlow unfortunately does not provide any tools for analysis or monitoring of workflows. Instead of a system that *runs* workflow, WebWorkFlow creates a system that is *workflow-enabled*.
- The current WebWorkFlow compiler is not yet up to industrial standards. The language still lacks certain features that should not be missing, would the technology be used in the industry. However, this is rapidly changing, because WebDSL is under constant development. For instance, during the course of this project, validation possibilities have been added to WebDSL.
- If a model is changed, recompiled and redeployed, the data model is likely to change, as well. This presents problems if one wants to change the application, but keep the data the current version is working with.

11.3 Future work

The workflow field of research is a large field, as is the web application domain. From the point of view of both domains, WebWorkFlow lacks certain features or has some disadvantages. This leads to manifold opportunities for future work. We will mention some of these here.

11.3.1 Timed Activities

Currently, all activities are either directly or indirectly - in the case of automated procedures - executed following user input. Many workflow applications however, contain a time aspect. This time aspect can be relative, for instance when a workflow task must be done within two weeks from a certain moment, or otherwise a reminder must be sent. The time aspect can also be absolute. Consider for example the case when a company wants to give their employees a small gift on their birthday. It would be handy to have a workflow activity warn a specific employee some time in advance to make sure the present is bought on time.

WebDSL does not currently support any scheduled execution of code. If support for this was added to WebDSL, we could extend WebWorkFlow with constructs that use these abstractions, to introduce both absolutely timed and relatively timed activities.

11.3.2 Web Services

The current version of WebWorkFlow is focused on building monolithic web applications. Many workflow systems, however, operate in the context of heterogeneous, autonomous, and/or distributed systems [22]. For WebWorkFlow to be able to function in this way, abstractions and functionality must be added to communicate with other systems. Web services could serve this goal. By adding support for web services to WebDSL, WebWorkFlow could fulfil two different roles in a workflow enactment system that surpasses the borders of an application generated by WebWorkFlow alone:

- As *orchestrator* of a group of workflow systems that together form a heterogeneous, distributed system. For this end, WebWorkFlow must be able to *speak* languages for web services, such as SOAP, for telling other systems what to do. This is a role that a BPEL workflow system fulfils. If this web service support is added to WebWorkFlow, a WebWorkFlow system can replace a BPEL system in real-life situations.

- As *participant* in a workflow system, together with other workflow systems, orchestrated by another party. For this end, WebWorkFlow must be able to *listen* to commands issued using a web service, in a language such as SOAP. In this case, WebWorkFlow systems can be used to implement part of the process.

The question is: how would the addition of web service support to WebDSL affect the WebWorkFlow language? In our opinion, adding web services to WebDSL would hardly influence the workflow language and can be considered as a mostly orthogonal issue. If WebDSL functions could be made available for calling from external systems using web services, the following scenario provides a way for WebWorkFlow applications to function as both an *orchestrator* and a *participant*:

- *Orchestration* of a group of workflow systems that together form a heterogeneous, distributed system can be implemented in the following way. If external web services can be called from WebDSL action code, it is possible to start a workflow using the `do {}` part of a procedure. By making available `processed()` as a function that can be called using a web service, it is possible to resume execution as soon as this function is called by the external system, indicating that it has finished with its work. The only change in the WebWorkFlow sub language would be providing a way to indicate that an externally visible function should be used instead of `processed()`.

  A potential problem with this solution, however, is how to handle any shared data. It might be feasible to implement the passing of objects between WebWorkFlow applications and external systems, but this would present problems in case the workflow exhibits parallel paths at the moment external systems are used, because data can be changed by both the external and internal procedures, in which case one of the changes is overwritten by the other.

- If a WebWorkFlow was to be used as *participant* in a workflow system, together with other workflow systems, orchestrated by another party, the following scheme would be a potential solution. If functions can be made available for calling by external systems using web services, it is possible to define a function that starts a workflow. On completing a workflow, it is possible to call a designated callback function defined at the workflow orchestrator.

A prerequisite for this solution is that the orchestrating system is able to perform tasks in this asynchronous way.

Both the orchestration and participation schemes use an asynchronous model for calling external functions, with a callback function to hand over control to the caller.

11.3.3 Workflow Variable Scope

As discussed in section 8.3, WebWorkFlow does not currently support any of the data patterns. This is caused by the fact that WebWorkFlow procedures do not have their own variable scope and means of passing data between different workflow elements. WebWorkFlow would benefit from a way of declaring variables for use in the scope of execution of a procedure, and ways of passing data from one procedure to another, or from parent procedures to child procedures and vice versa.

11.3.4 Separating UI from Procedures

The issue concerning separation of UI and procedure code that was raised in section 7.5.3 is a disadvantage from the workflow specification point of view. A possible solution for this problem, however, is introduced in the same section.

11.3.5 Generated Task Lists

As mentioned in section 7.5.3, the task lists that are generated for workflows do not give the user the workflow overview that we want. It would be good to extend the WebWorkFlow compiler so that task lists can be generated including the following: displaying previously performed procedures, displaying tasks that can be performed by other actors, and displaying tasks that will become available in the future.

11.3.6 Direct Support for Multiple Instances

At the moment, all patterns involving multiple instances are implemented using a workaround that involves explicitly defining a subcase entity and defining procedures for that entity instead of the original entity. This is a rather elaborate way of using instances in workflows. It might be good to extend WebWorkFlow to include syntax for instances so these patterns can be implemented more simply.

On the other hand, in our opinion, running multiple instances of exactly the same workflow on the same object is not very useful. When multiple instances of tasks for the same object are needed, this usually is for allowing multiple actors to perform the same task for that object. But the same task performed by multiple actors are in fact not identical, as they differ in respect to the actor. Adding an explicit instance entity that captures the performing actor in an instance-specific property is therefore a good idea.

11.3.7 PDF Generation

As shown in chapter 7, some applications use an algorithm to generate PDF documents from some input or application data. Adding this to the WebWorkFlow implementation of the LIXI valuation workflow was not possible yet. PDF generation abstractions could be added to WebDSL.

11.3.8 Support for Testing, Analysis and Monitoring

So far, WebWorkFlow is only concerned with adding workflow support to web applications. Unfortunately, it does not provide tools for testing, analysis, or monitoring of workflows. Because the control-flow descriptions in WebWorkFlow are much less isolated than in other workflow solutions, as both procedure-level amendments and normal WebDSL code can influence execution of processes, it will not be easy to create tools for these purposes. However, it might be worth looking into this.

11.3.9 Transactions

Using transactions, it is possible to securely perform parts of a workflow or the whole workflow, or rollback any changes in case the workflow cannot be successfully completed. Transactions can also be used in other situations. WebDSL does not yet have transaction support.

11.3.10 Automatic Database Migration

As already mentioned in section 11.2.2, change of the model implies change of the data model, which can present problems. This problem needs further investigation. A start with this was made in [67].

11.3.11 Optional Values for Entity Properties

At present, it is not possible to use optional values for value properties of entities in WebDSL. This results in problems if data fields are not filled in, or in automatic insertion of zeroes when integer fields are left empty. This is an issue that needs fixing.

Bibliography

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K Liu, D. Roller, D. Smith, Satish Thatte, I Trickovic, and S. Weerawarana. Business process execution language for web services, version 1.1. Technical report, BEA, IBM, Microsoft, SAP, Siebel, 2003.

[2] Thomas Barothy, Markus Peterhans, and Kurt Bauknecht. Business process reengineering: emergence of a new research field. *SIGOIS Bull.*, 16(1):3–10, 1995.

[3] Tim Berners-Lee, Robert Cailliau, Jean-Franois Groff, and Bernd Pollermann. World-wide web: The information universe. *Internet Research*, 2:52–58, 1992.

[4] Egon Börger. High level system design and analysis using abstract state machines. In *FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method*, pages 1–43, London, UK, 1999. Springer-Verlag.

[5] Marco Brambilla, Stefano Ceri, Piero Fraternali, and Ioana Manolescu. Process modeling in web applications. *ACM Trans. Softw. Eng. Methodol.*, 15(4):360–409, 2006.

[6] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software – Practice & Experience*, 30:259–291, 2000.

[7] M.G.J. van den Brand and P. Klint. Aterms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology*, 49(1):55–64, 2007.

[8] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 2008. Special issue on Experimental Systems and Tools.

[9] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2004. ACM.

[10] Franck van Breugel and Maria Koshkina. Models and verification of bpel, 2006. http://www.cse.yorku.ca/franck/research/drafts/tutorial.pdf, 2006.

[11] Laura Bucci. Organizing an acm/sig conference, August 2008. http://www.acm.org/sigs/.

[12] Stefano Ceri, Piero Fraternali, and Maristella Matera. Conceptual modeling of data-intensive web applications. *IEEE Internet Computing*, 6(4):20–30, July 2002.

[13] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. Technical report, W3C, March 15, 2001.

[14] Thomas H. Davenport and James E. Short. The new industrial engineering: Information technology and business process redesign. *Sloan Management Review*, pages 11–27, Summer 1990.

[15] J. Eder, H. Groiss, and W. Liebhart. The Workflow Management System Panta Rhei. *Advances in Workflow Management Systems and Interoperability. Springer, Istanbul, Turkey, August*, pages 129–144, 1998.

[16] C. Ellis and G. Nutt. Workflow: The process spectrum, 1996.

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

[18] D. Fahland and W. Reisig. Asm-based semantics for bpel: The negative control flow. In *Proceedings of the 12th International Workshop on Abstract State Machines*, pages 131–151, March 2005.

[19] R.T. Fielding and R.N. Taylor. Principled design of the modern web architecture. *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 407–416, 2000.

[20] Jesús Arias Fisteus, Luis Sánchez Fernández, and Carlos Delgado Kloos. Applying model checking to bpel4ws business collaborations. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 826–830, New York, NY, USA, 2005. ACM.

[21] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting bpel web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM.

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

[23] Danny Groenewegen and Eelco Visser. Declarative access control for WebDSL: Combining language integration and separation of concerns. In Daniel Schwabe and Francisco Curbera, editors, *International Conference on Web Engineering (ICWE 2008)*. IEEE CS Press, July 2008.

[24] Michael Hammer. Reengineering work: Don't automate, obliterate. *Harvard Business Review*, pages 104–112, July-August 1990.

[25] Z. Hemel, L. C. L. Kats, and E. Visser. Code generation by model transformation. A case study in transformation modularity. In J. Gray, A. Pierantonio, and A. Vallecillo, editors, *International Conference on Model Transformation (ICMT08)*, Lecture Notes in Computer Science. Springer, June 2008.

[26] R.A. Hirschheim. Office automation: a social and organisational perspective. Wiley, 1985.

[27] David Hollingsworth. The workflow reference model. Technical report, Workflow Management Coalition, 1995.

[28] G.J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, May 1997.

[29] Thomas Hornung, Agnes Koschmider, and Jan Mendling. Integration of heterogeneous bpm schemas: The case of xpdl and bpel. In *CAiSE Forum*, 2006.

[30] Kurt Jensen, Lars Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3):213–254, June 2007.

[31] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, AT&T, 1975.

[32] Mikko Kontio. Architectural manifesto: Mda in action, October 2005. http://www-128.ibm.com/developerworks/library/wi-arch19/.

[33] Dongsheng Liu, Jianmin Wang, Stephen C. F. Chan, Jiaguang Sun, and Li Zhang. Modeling workflow processes with colored petri nets. *Comput. Ind.*, 49(3):267–281, 2002.

[34] LIXI. Lixi website, December 2008. http://www.lixi.org.au/about.html.

[35] Dirk E. Mahling, Noël Craven, and W. Bruce Croft. From office automation to intelligent workflow systems. *IEEE Expert: Intelligent Systems and Their Applications*, 10(3):41–47, 1995.

[36] Dragos A. Manolescu. Workflow enactment with continuation and future objects. *SIGPLAN Not.*, 37(11):40–51, 2002.

[37] John McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962.

[38] S. McCready. There is more than one kind of workflow software. *Computerworld*, November 2, 1992.

[39] K. L. Mcmillan. The smv system - for smv version 2.5.4, 2000.

[40] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the .net framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, New York, NY, USA, 2006. ACM.

[41] M. Mernik, J. Heering, and A. M. Sloane. When And How To Develop Domain-Specific Languages. cwi, CWI, 2005. bibliographical data to be processed – CWI. Software Engineering [SEN] ; E 0517. ISSN: 1386-369X – CWI –.

[42] John A. Miller, Devanand Palaniswami, Amit P. Sheth, Krys J. Kochut, and Harvinder Singh. Webwork: Meteor$_2$‘s web-based workflow management system. *J. Intell. Inf. Syst.*, 10(2):185–215, 1998.

[43] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[44] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[45] Object Management Group. *Business Process Modeling Notation Specification, OMG Final Adopted Specification*, 1.0 edition, February 2006.

[46] Margrethe H. Olson and Henry C. Lucas, Jr. The impact of office automation on the organization: some implications for research and practice. *Commun. ACM*, 25(11):838–847, 1982.

[47] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.

[48] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[49] Frank Puhlmann and Mathias Weske. Using the π-calculus for formalizing workflow patterns. pages 153–168. 2005.

[50] J. Recker and M. Strategy. Process Modeling in the 21 stCentury. *BPTrends, May*, pages 1–8, 2006.

[51] Tom Rodden. A survey of cscw systems. *Interacting with Computers*, 3:319–353, 1991.

[52] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow resource patterns.

[53] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns. In *Proc. of 24th Int. Conf. on Conceptual Modeling (ER05)*, pages 353–368, 2005.

[54] Nick Russell, Arthur, Wil M. P. van der Aalst, and Natalya Mulyar. Workflow control-flow patterns: A revised view. Technical report, BPMcenter.org, 2006.

[55] Nick Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and Natalya Mulyar. Workflow control-flow patterns: A revised view. Technical report, BPMcenter.org, 2006.

[56] Nick Russell, Wil M. P. van der Aalst, and Arthur H.M. ter Hofstede. Exception handling patterns in process-aware information systems. Technical report, BPMcenter.org, 2006.

[57] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[58] The AndroMDA Team. *AndroMDA BPM4Struts Cartridge*, 3.2 edition, 2003. http://galaxy.andromda.org/docs-3.2/andromda-bpm4struts-cartridge/index.html.

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

[60] W. M. P. van der Aalst. Three good reasons for using a Petri-net-based workflow management system. In S. Navathe and T. Wakayama, editors, *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96),*, pages 179–201, 1996.

[61] W. M. P. van der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[62] Wil M. P. van der Aalst. Patterns and XPDL: A critical evaluation of the XML process definition language. Technical report, BPMcenter.org, 2003.

[63] Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, Nick Russell, H. M. W. (Eric) Verbeek, and Petia Wohed. Life after bpel? In *EPEW/WS-FM*, pages 35–50, 2005.

[64] Wil M. P. van der Aalst, Jens Bk Jrgensen, and Kristian Bisgaard Lassen. Let's go all the way: From requirements via colored workflow nets to a bpel implementation of a new bank system. In Robert Meersman, Zahir Tari, Mohand-Said Hacid, John Mylopoulos, Barbara Pernici, zalp Babaoglu, Hans-Arno Jacobsen, Joseph P. Loyall, Michael Kifer, and Stefano Spaccapietra, editors, *OTM Conferences (1)*, volume 3760 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 2005.

[65] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[66] A. van Deursen, P. Klint, and J. M. W. Visser. Domain-Specific Languages. cwi, CWI, 2000. CWI. Software Engineering [SEN] ; R 0032. ISSN: 1386-369X – CWI – 13 p.

[67] Sander D. Vermolen and Eelco Visser. Heterogeneous coupled evolution of software languages. *Lecture Notes in Computer Science*, 5301:630–644, September 2008. In K. Czarnecki and I. Ober and J.-M. Bruel and A. Uhl and M. Voelter (eds.) Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008). Toulouse, France, October 2008.

[68] Eelco Visser. A family of syntax definition formalisms. In M. G. J. van den Brand et al., editors, *ASF+SDF 1995. A Workshop on Generating Tools from Algebraic Specifications*, pages 89–126. Technical Report P9504, Programming Research Group, University of Amsterdam, May 1995.

[69] Eelco Visser. Domain-specific language engineering. Reflections on the design and implementation of domain-specic languages with a case study in the domain of web engineering. Presentation in the TU Delft course on Program Transformation and Generation, May 2008.

[70] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008.

[71] David Wastell and Phil White. Using process technology to support cooperative work: Prospects and design issues. In *CSCW in Practice: An Introduction and Case Studies*, pages 105–126. Springer-Verlag, 1993.

[72] S.A. White. Introduction to BPMN. *IBM Cooperation*, July 2004.

[73] Stephen A. White. Using bpmn to model a bpel process., March 2005.

[74] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell. On the suitability of bpmn for business process modelling. In Schahram Dustdar, Jos Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2006.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

Automated Task: A *task* that is performed by a computer, without human intervention.

Activity: See *task*

Actor: See *participant*

Agent: See *participant*

BPEL: Short for BPEL4WS

BPEL4WS: Business Process Execution Language for Web Services, see section 9.5

BPM: Business Process Modeling

BPR: Business Process Reengineering

Business Process: A recurrent pattern of tasks performed in the interest of reaching a business goal.

CSCW: Computer-Supported Cooperative Work

HAD System: Heterogeneous, Autonomous and/or Distributed System

Office Automation: "The use of integrated computer and communication systems to support administrative procedures in an office environment" [46, 71]

OIS: Office Information System

Participant: Either specific persons, or persons that have a specific *role*, like administrator, valuer or manager, that participates in a workflow, or a software system that acts as a workflow participant

Process: See *Business Process*

Process Definition, Process Description: The actual ordering of events is directed by the description of the process. It also specifies conditions under which the tasks are to be performed

Procedure (in general): See *Business Process*

Procedure (WebWorkFlow): An atomic unit of work, or a composition of other procedures, together with a *process definition*

Resource: Either human workers, or non-human resources, such as equipment or computers

Task: Can be an atomic activity such as filling in a form, sending an email, or handing over a key. Tasks can also consist of groups of tasks, following a process description: a *subworkflow*.

WfMC: Workflow Management Coalition

WFMS: Workflow Management System

Workflow: A *workflow* consists of *tasks*, performed by *participants* to accomplish some business goal. In addition to this, a workflow contains a *process description*, defining the order of task invocation, condition(s) under which tasks must be invoked and to what participants the tasks must be delegated.

Workflow Management: A technology supporting the initial design and reengineering of both business and information processes. It involves defining workflows, and providing for fast (re)design and (re)implementation of the processes as business needs and information systems change [22]

Workflow Reference Model: A reference architecture for WFMSs, developed by the WfMC

XPDL: The XML Process Definition Language, developed by the WfMC as a reference process definition language