Workflow Control for Distributed Natural Language Processing Applications

Wang Ling

Engenharia Informática e de Computadores

Júri
Presidente: Professor Doutor Nuno João Neves Mamede
Orientador: Professor David Manuel Martins de Matos
Vogais: Professora Doutora Maria Luísa Torres Ribeiro Marques da Silva Coheur
Professor Doutor Luís Manuel Antunes Veiga

Outubro de 2009
Resumo

Neste trabalho, apresentamos um sistema de gestão de fluxos de trabalho que permite a modelação, execução e a monitorização de fluxos de trabalho orientados ao campo do Processamento da Língua Natural (PLN), cujo é um campo que trabalha com grandes quantidades de poder de processamento. Este campo impõe requisitos sobre as quais construímos o nosso sistema. Estes requisitos incluem (1) usar uma língua de fácil aprendizagem, até para indivíduos com pouca experiência em programação, e ao mesmo tempo ter a expressividade para permitir a modelação de fluxos de trabalho complexos, (2) suporte a ambientes em grid para componentes que processam grandes quantidades de informação e extensibilidade para permitir adição de módulos para trabalhar com novos ambientes, (3) tolerância a erros permitindo a recuperação de um erro sem reiniciar o fluxo e (4) facilidade de efectuar alterações num fluxo de trabalho.

O sistema de gestão de fluxos de trabalho que este trabalho apresenta é implementada numa arquitectura cliente-servidor, no qual, depois de montar o servidor, os utentes acedem-no a partir de uma interface browser. O sistema permite a especificação de fluxos de trabalho através de uma interface gráfica onde os componentes podem ser interligados, criando um fluxo de trabalho com pouca necessidade de escrever linhas de código.
Abstract

In this work, we present a workflow management system that allows the modelling, execution and monitoring of workflows oriented to the field of Natural Language Processing (NLP), which is a field that requires large amounts of processing power. This field imposes requirements, which we build our system to answer. These requirements include (1) using a language that is simple to learn, even for non-programmers, but at the same time possesses the expressiveness to enable the modelling of complex workflows, (2) support for grid environments for components that need to process large amounts of data, and the extensibility to allow the addition of modules to work with other environments, (3) error tolerance that allows recovering an error without restarting the workflow and (4) the ability to easily make changes to a workflow.

The workflow management system that this work presents is implemented using a client-server architecture, where, after deploying the server, the users access it using a browser interface. The system allows the specification of workflows through a graphical interface where components can be linked together to create a workflow with little need for coding.
Palavras Chave

Workflow
Processamento de Lingua Natural
YAWL
Grid Computing
High Performance Computing

Keywords

Workflow
Natural Language Processing
YAWL
Grid Computing
High Performance Computing
Acknowledgements

I would like to show my appreciation to my supervisor David Matos, for providing support and guidance during this work.

I would also like to thank the YAWL development team for providing me with answers when I needed them.

Finally, I am indebted to all my friends, family and the staff in NLP for being there for me.

Lisboa, October 23, 2009

Wang Ling
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problems</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Goals</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Structure of this Document</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Workflow Management System Model</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Application Interface</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1 Web Services</td>
<td>7</td>
</tr>
<tr>
<td>2.1.2 Condor</td>
<td>8</td>
</tr>
<tr>
<td>2.1.3 Condor-G</td>
<td>8</td>
</tr>
<tr>
<td>2.1.4 Summary</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Workflow Modeling Languages</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Workflow Patterns</td>
<td>9</td>
</tr>
<tr>
<td>2.2.2 Petri Nets</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3 High-Level Petri Nets</td>
<td>10</td>
</tr>
<tr>
<td>2.2.4 Workflow nets</td>
<td>11</td>
</tr>
<tr>
<td>2.2.5 YAWL Language</td>
<td>12</td>
</tr>
<tr>
<td>2.2.6 UML Activity Diagrams</td>
<td>13</td>
</tr>
<tr>
<td>2.2.7 BPEL4WS</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Comparison Between Modeling Languages</td>
<td>13</td>
</tr>
<tr>
<td>2.3.1 Expressiveness</td>
<td>13</td>
</tr>
<tr>
<td>2.3.2 Analyzability</td>
<td>14</td>
</tr>
<tr>
<td>2.3.3 Usability</td>
<td>15</td>
</tr>
<tr>
<td>2.3.4 Support</td>
<td>16</td>
</tr>
</tbody>
</table>
2.3.5 Summary .................................................. 16

2.4 Workflow Engines ......................................... 17

2.4.1 DAGMan .................................................. 17

2.4.2 P-grade ................................................... 18

2.4.3 BPEL4WS .................................................. 18

2.4.4 YAWL ....................................................... 19

2.5 Comparison Between Workflow Engines .................. 19

2.5.1 Expressiveness ........................................... 19

2.5.2 Environment ............................................. 19

2.5.3 Summary .................................................. 20

3 YAWL architecture ............................................. 21

3.1 YAWL engine .................................................. 21

3.2 Custom Services ............................................. 21

3.3 YAWL specification ......................................... 22

3.3.1 “WebServiceGatewayFactsType” decomposition .......... 24

3.3.2 “NetFactsType” decomposition .......................... 24

3.3.2.1 “Task” element ..................................... 25

3.4 Resourcing Service ......................................... 28

3.4.1 User Model .................................................. 28

3.4.2 Resourcing Specification ................................. 28

3.5 Summary ..................................................... 30

4 System Architecture .......................................... 31

4.1 Custom Services ............................................. 31

4.2 Workflow Model ............................................. 33

4.3 Workflow Server ........................................... 36

4.3.1 Database Gateway Layer ................................. 37

4.3.2 Marshaller .................................................. 37

4.3.3 YAWL Interface Client .................................. 38
4.3.4 Service Callback Interface ........................................... 38
4.3.5 Manager ................................................................. 39
4.3.6 Server Interface ....................................................... 39
4.3.7 Web Container .......................................................... 40

4.4 User Interface ............................................................. 40
4.4.1 GWT toolkit ............................................................. 41
4.4.2 Animation Engine ....................................................... 41
4.4.3 Server Interface Client ................................................ 42
4.4.4 Interface Elements ..................................................... 42
4.4.5 Desktop ................................................................. 43
4.4.6 Task Queue .............................................................. 43
4.4.7 Item Search ............................................................. 43
4.4.8 Workflow Viewer ....................................................... 44
4.4.9 Workflow Editor ........................................................ 44
4.4.10 Interface Manager .................................................... 46

4.5 Summary ................................................................. 46

5 System Implementation .................................................... 49
5.1 YAWL conversion .......................................................... 49
5.1.1 Node Expansion ......................................................... 49
5.1.2 Conversion to YAWL Specification .................................. 50

5.2 Workflow Importing ....................................................... 51
5.2.1 Workflow Node ........................................................ 51
5.2.2 Workflow Specification Injection ..................................... 51
5.2.3 Workflow Versioning .................................................. 51

5.3 Workflow Monitoring ..................................................... 52
5.3.1 Event Logging .......................................................... 52
5.3.2 EventLogger Service .................................................. 53
5.3.3 EventLogger Node ...................................................... 53
5.4 Condor Support ........................................ 56
5.4.1 BirdBath ........................................ 56
5.4.2 Condor Service ................................. 56
5.4.3 Condor Nodes ................................. 57
  5.4.3.1 Submitting Jobs .......................... 57
  5.4.3.2 Querying Jobs ............................ 57
  5.4.3.3 Fetching The Output ..................... 57
5.4.4 Job Monitorization ............................. 58
5.4.5 Error Tolerance ............................... 58
5.5 User Interaction ................................ 59
  5.5.1 YAWL Resource Service ................. 59
  5.5.2 Creating users .............................. 59
  5.5.3 Completing Jobs ............................ 60
  5.5.4 Allocating Jobs ............................ 60
5.6 Other Services .................................. 61
  5.6.1 Web Services Support ..................... 61
  5.6.2 HTTP Support .............................. 61
  5.6.3 Scheduling Jobs ............................ 61
5.7 Extending Services ............................. 62
5.8 Summary .......................................... 62

6 Evaluation .......................................... 65

7 Conclusion .......................................... 67
  7.1 Future Work ..................................... 68
    7.1.1 Permission and user management ........ 68
    7.1.2 Workflow Validation ....................... 68
    7.1.3 Implementing Additional YAWL Services . 68
    7.1.4 Support for BPEL .......................... 68
    7.1.5 Improve Workflow Monitoring Interface .. 69
A Usage Example

A.1 Logging in ................................................. 77
A.2 Workflow Definition .................................... 77
  A.2.1 Creating a new workflow ......................... 77
  A.2.2 Adding a root net .................................. 77
  A.2.3 Adding new nodes .................................. 81
  A.2.4 Adding new variables ............................... 81
  A.2.5 Setting parameter mappings ....................... 81
  A.2.6 Saving the workflow ............................... 85
  A.2.7 Executing the workflow ......................... 85
A.3 Summary .................................................... 85

B Installation Guide .......................... 89

B.1 Installation Instructions .......................... 89
B.2 Configuration After Installation .................. 89
B.3 Installing from source code ...................... 92
B.4 Importing projects into Netbeans ................ 92
B.5 Creating Users ......................................... 95
B.6 Installing the YAWL Custom Services .......... 95
# List of Figures

2.1 Workflow Management System Model .......................... 6  
2.2 Classic Petri Net and Colored Petri Net Comparison ................. 11  
2.3 Process modeled in a workflow net ................................ 12  
3.1 SpecificationSet Element Example ................................ 23  
3.2 Specification Element Example .................................. 24  
3.3 “WebServiceGatewayFactsType” Decomposition Example .............. 25  
3.4 “NetFactsType” Decomposition Example .......................... 26  
3.5 Task Element Example ............................................. 27  
3.6 Resoucing Element Example ....................................... 29  
4.1 Architecture Overview ............................................ 32  
4.2 Server Side Architecture .......................................... 32  
4.3 Workflow Model .................................................... 35  
4.4 Node Types .......................................................... 35  
4.5 Interface Architecture .............................................. 41  
4.6 Desktop ............................................................. 43  
4.7 Workflow Editor Interface ......................................... 45  
4.8 Interface Manager Communication protocol ......................... 47  
5.1 Web Service expansion with error-handling ......................... 50  
5.2 Logging workflow nodes ........................................... 54  
5.3 Logging workflow nodes without a wrapper net .................... 55  
5.4 Condor Job Monitoring ............................................. 58  
5.5 Parameter specification XML example ............................ 61
List of Tables

2.1 Comparison between HLPN, YAWL, UML and BPEL4WS . . . . . . . . . . 14
Introduction

Natural Language Processing is a branch of artificial intelligence that studies the process of human language understanding and automated generation. The goal is to get computers to perform useful tasks involving human language [39]. This field is similar to e-science fields, such as astronomy and gravitational wave science, where large amounts of data must be processed.

One approach to compensate for this is the use of grid computing, enabling collaborative use of resources to raise system performance. This is essentially a divide-and-conquer approach, where, for example, one large file with 3 Gigabytes can be divided into 6 smaller files with 500 Megabytes each, which are processed by 6 different CPUs.

Another way to boost performance is to optimize the interaction between components. Usually, NLP systems are created from the integration of a large number of different stand-alone components that perform simpler tasks, such as part-of-speech tagging, information extraction, or word sense disambiguation. By optimizing the sequence in which those components are invoked during the execution of an instance of the system, we can further improve the performance of the system; for example, take a system with 3 components (A, B and C) and say that B’s input was the output of A. Normally, we would invoke A, B, then C. However, component C does not need to wait until the other components finish before starting. By invoking A and C simultaneously on different CPUs, the system’s performance can be improved. Furthermore, this approach can be combined with the strategy above by running each or a subset of the components A, B, and C in separate CPUs.

The former generally requires a grid computing environment, where available work is managed and distributed to the available resources. This is generally done using workload management systems such as Condor [57]. The latter can be achieved by creating and synchronizing multiple processes/threads so that the system progresses accordingly to the dependences defined between the components within the system. This task is easy when the system is only composed by a small amount of components with few dependencies between them but the complexity escalates rapidly as systems grow larger. As a means to make this process simpler, workflows are often used.

A workflow can be defined as a series of coordinated computational tasks according to a set
of procedural rules to deliver a service. A workflow model is the definition of workflow and its instantiation is denominated a case. The actual execution of these cases is handled by a workflow management system. A useful property of workflows is that they do not need to know about the applications that execute the tasks, thus allowing subsequent changes to be incorporated into those applications without compromising the integrity of the workflow. Finally, workflows are a particularly useful way to model systems, providing users a high-abstraction view of the system as a whole and how the system evolves during its execution.

Furthermore, it is convenient to separate the development and the execution of applications into different concerns [18]. At the development-level, the application can be modelled independently of the execution environment as abstract workflows. However, at the execution-level, the individual tasks need to be implemented and assigned an execution environment.

1.1 Problems

Most workflow management systems are aimed at providing services to a general public rather than a specialized one. Because of this, some problems may arise that are specific to the area of NLP.

- Each of the NLP components generally takes years to develop, and for that reason, they are usually implemented by independent research groups, who are more focused in their particular projects rather than on the whole system. As a result, when integrating these components, compatibility issues are bound to emerge. Those components might be working under distinct platforms and be implemented in programming languages, making the programmer go through the unnecessary trouble of writing additional pieces of code to integrate each pair of inter-communicating components. To make matters worse, NLP components are rarely used by one particular project exclusively, but are frequently implemented with a tight coupling to the particular project that they were made for. As a consequence, it becomes harder for them to be reused in other projects. The answer to this problem for most workflow management systems is to provide a heterogeneous service, such as Web Services. However, most developers in NLP are not familiar with these technologies, and do not want to adapt their scripts to allow their invocation by the workflow management system.

- Another problem lies in the fact that, in the field of NLP, as in many other e-science fields, not every researcher is necessarily a programmer. Some of them can be, for instance, from a linguistic background and have little programming skills. In contrast, mainstream workflow solutions on the market, such as BPEL4WS, are oriented for
programmers and have a steep learning curve making them not the best choice for developing NLP applications.

- Very few workflow management systems available provide support for grid computing, since high-throughput computing is not required for most industries. However, in NLP, this is considered a requirement, since single components can take hours or days to finish without it.

### 1.2 Goals

The purpose of this project is to build a workflow management system capable of leveraging the potential of the workflows to provide a more agile development of NLP applications. This will be achieved by accomplishing the following sub-goals:

- **Promote the workflow paradigm:** the framework must allow the creation of workflows and their execution. Moreover, they must be flexible, allowing different configurations build with little effort.

- **Improve the intrinsic-interoperability of components:** the NLP components must be able to communicate easily with other components, regardless of the programming language and the platform they are implemented in.

- **Support for grid computing:** the framework must bridge the workflow management and grid environments.

- **Separation the roles within the organization:** the system must draw the boundary between the programmers and the software engineers, much like in SOC (service oriented-computing) [20]. While the former are focused in particular components, the latter use those to build a system.

- **Provide a graphical interface that is easy to learn and use, for non-programmers and programmers alike.**

### 1.3 Structure of this Document

This document is structured as follows:

- Chapter 2 will provide the general architecture of a workflow management system. Then, each of the main components of this architecture is analyzed and many solutions are compared in order to find which would be more suited to this work.
• Chapter 3 will give an overview of the architecture of the workflow engine that we will be using called YAWL (Yet Another Workflow Language), where as, chapter 4 will provide the architecture of our workflow management system.

• Chapter 5 describes the implementation details of the main features of our system.

• Chapter A will illustrate how the interface is used, by giving a example of the definition and execution of a simple workflow.

• Chapter 6 evaluates our system in terms of strengths and weaknesses.

• Chapter 7 ends this work with the presentation the conclusions and future work.
A workflow can be defined as a series of coordinated computational tasks according to a set of procedural rules to deliver a service. A workflow model is the definition of workflow and its instantiation is denominated a case. In grid workflows, each task within the workflow is frequently executed on different environments and distributed across various systems. A useful property of workflows is that they do not need to know about the applications that execute the tasks, thus allowing subsequent changes to be incorporated into those applications without compromising the integrity of the workflow.

Before going into details, it is essential that we define the model of our system, delineating the different components required for its assembling. We will base our system on the model proposed by David Hollingsworth [34], which is illustrated in Figure 2.1.

When designing a workflow, a user specifies processes, which are comprised by the tasks that need to be done, as well as their ordering. This specification is done with process definition tools. The most important aspect in these tools is the language, which will be used to model the workflows, since different languages possess different sets of properties. Thus, we will dedicate section 2.2 to the analysis of the approaches taken by different languages. Because the modeling language is the interface users will be using to design their applications, the main concerns we will be taking into account are:

- **Expressiveness**: What kind of constructs are allowed by the language to model a workflow: e.g. conditional routing, parallel execution and looping.
- **Analyzability**: Whether the workflows can be analyzed for correctness at design time.
- **Usability**: The ease of learning and designing workflows with the language.
- **Support**: Are there many tools that support the language.

After workflows are designed, they need to have an execution environment where they can be executed. This is done by the workflow enactment service, where a workflow definition can be submitted, managed, and executed. The actual execution of the process is delegated to a workflow engine. There are many workflow engines in the market today that can be used.
CHAPTER 2. WORKFLOW MANAGEMENT SYSTEM MODEL

Figure 2.1: Workflow Management System Model

The most relevant ones will be presented in section 2.4. Workflow engines generally have their own language to specify workflows. However, these will most likely be transparent to the user, which is why user-centered properties like usability will not be taken into account. Another aspect that will not matter is the performance of the engine, since the execution time for NLP components can vary between a few minutes and several hours or even days. Thus, it is unlikely that the time an engine needs to coordinate the tasks will influence the overall performance of an application. The main aspects we will discuss will be:

- **Expressiveness**: Although users will not be using the language directly, our framework will have to map the modeling language to the engine language. The mapping of less expressive languages to more expressive ones is generally not trivial and sometimes impossible.

- **Environment**: How does the engine communicate with the surrounding environment.

During the execution of the workflow, external entities may be invoked. Invocations can occur either via the client application’s interface or the invoked application’s interface. The main difference between the two is that the former is oriented to work with human resources and the latter is focused on computational tasks. We will discuss possible interfaces for invoked applications in section 2.1. We will not talk about the client interfaces in this work, since NLP applications rarely involve human tasks in their workflows.

Finally, the administration and monitoring tools, and other workflow enactment interfaces allow the system to cooperate with other workflow management systems from other vendors.
by defining a standard interface. We will also omit these components, as it is not part of our goal to create a system capable of interacting with other workflow management systems.

2.1 Application Interface

The Application Interface is the bridge between the workflow manager and external entities. This section is not intended to make a comparison between different interfaces but rather to give an overview of the different interfaces our framework should support. The main requirements the application interface in our system is that our set of external entities supports grid computing and offers a large variety of components to use. An example of the former is Condor [57], and Web Services [66] are an example for the latter.

2.1.1 Web Services

A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. The Web Service [52] framework can be described in terms of:

- The protocols and formats that can be used to access services and to communicate with them, such as SOAP (Simple Object Access Protocol) [50].
- The language the services are described, such as WSDL (Web Service Description Language) [15].
- The way the services are discovered, such as UDDI (Universal Description Discovery and Integration) [16].

The advantages of using Web Services include:

- Loose coupling - Web services only publish their interface and functions, which means that clients can invoke its services while remaining oblivious to the implementation of the service.
- Widely Supported - Large amount of middleware and toolkits exist that support development with Web Services.

One problem with Web Services is the fact that they can be overly complex. Although many languages, such as Java possess toolkits that simplify the process of defining new Web Services to overcome this problem, these attempts raise other problems. One of these is the fact that WSDL documents that are automatically generated are often too hard to read. Another problem is the fact that developers lose control over the WSDL document since any change in the server application has a chance of resulting in a completely different WSDL.
2.1.2 Condor

Condor is a specialized job and resource management system for compute intensive jobs [57]. It provides a job management mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Upon submitting a job, Condor allocates the resources to run it based on a policy, monitors its progress and informs the user upon completion. Condor distances from other resource management systems as it provides high-throughput computing and opportunistic computing environments. A high-throughput computing environment can provide with large amounts of fault-tolerant processing capacity over long periods of time through the effective management and exploitation of available resources [45]. The opportunistic computing environment aims to fully use the available resources whenever they are available, endeavoring to keep resources from going idle.

2.1.3 Condor-G

Condor-G [27] is the marriage of the Condor Project with the Globus Project [24]. The Globus Toolkit provides security, resource discovery, and resource access in multi-domain environments while Condor manages the jobs and resources within an administrative domain [26]. This has the benefit of allowing users to use multi-domain resources as their own.

2.1.4 Summary

Web Services deal with the heterogeneity of applications that must interoperate with the workflow as NLP components are generally implemented in various different environments. Although having a Web Service interface allows the interoperation with all environments that have a Web Service interface, having a direct interface for Grid-related technologies such as Condor and Condor-G that are frequently used in NLP makes the life easier for users.

2.2 Workflow Modeling Languages

The workflow management taxonomy proposed by Jia Yu and Rajkumar Buyya [70] defines two main types of workflow modeling languages: graph-based and language-based. Graph-based modeling allows users to work with a representation of the workflow. Users compose workflows by inserting the elements of interest into the workflow. This type of modeling language enables users to work on a higher-level of abstraction. The main languages for graph-based modeling are PNs (Petri Nets) [62], YAWL (Yet Another Workflow Language) [33] and UML (Unified Modeling Language) Activity Diagrams [54]. Language-based modeling allows
users to specify the behavior of a workflow in terms of a language. The most commonly used type of language are markup languages such as XML [67]. This way of modeling systems become more accessible for experienced users, as they need to memorize the syntax of the language. Examples of languages that take this approach include, BPEL4WS (business process execution language) [44], WSFL (Web Services Flow Language) [43] and Xlang [17]. BPEL4WS [56] is the result of merging Microsoft’s Xlang specification with IBM’s WSFL, possessing features from both languages. Thus, the analysis of Xlang and WSFL will not be covered.

### 2.2.1 Workflow Patterns

Before talking about workflow languages, it is important to have a formal specification to classify a workflow’s expressiveness. The approach we take in this work is based on workflow patterns [1](these refer to table 2.1). This document describes a series 20 patterns through which the expressiveness of each language can be compared. The following gives a brief overview of those patterns:

- Patterns 1 to 5 describe the most basic constructs, which enable the modeling of sequential, parallel and conditional routing.
- Patterns 6 to 9 describe advanced branching/synchronization constructs where a specific number of branches can be chosen with various joining methods.
- Pattern 10 describes the constructs for loops.
- Pattern 11 describes the constructs for implicit termination, where a process terminates when there are no more activities to be done.
- Patterns 12 to 15 refer to the constructs that allow multiple instances of activities to be executed.
- Patterns 16 to 18 concern the specification of state-based behavior
- Patterns 19 and 20 refer to the cancelation of activities.

### 2.2.2 Petri Nets

One popular choice for modeling workflows are Petri Nets (PNs). A PN is a formalized mathematical modeling language commonly used to areas such as communication protocols, performance evaluation, and distributed systems. They have been proposed for modeling
workflows for a long time. Thus, a large amount of work was spent researching this subject. Some workflow management systems that are based on PNs include Grid-flow [30] and XRL/Flower [64]. CPNs (Classic PNs) can be viewed as directed bipartite graphs where nodes are either places or transitions. These nodes are connected through directed arcs and places can only connect to transitions and vice-versa. Places are generally represented by circles and transitions by rectangles. Formally, a is a triple \((P, T, F)\) [62], where:

- \(P\) is a finite set of places
- \(T\) is a finite set of transitions \((P \cap T = \emptyset)\)
- \(F \subseteq (P \times T) \cup (T \times P)\) is a set of arcs (flow relation).

A place \(p\) can be an input place for a given transition \(t\) if it has an arc directed to \(t\) and output place if \(t\) has an arc directed to \(p\). The set of input places of a given transition \(t\) is denoted by \(*t\) and the set of output places of \(t\) is denoted by \(t^*\). In the same sense, \(*p\) is used to express the transitions sharing \(p\) as output and \(p^*\) the transitions with \(p\) as an input. Each place \(p\) is associated with 0 or more tokens, drawn as dots. The state \(M\) of the net is defined as the distribution of these tokens by the set of places. When a transition \(t\) has tokens in each of its input places \((*t)\), they are considered enabled, and at any moment they may fire, consuming one token from each of the input places and producing a token in each of it’s output places \((t^*)\). This, in turn, changes the state \(M\) of the net.

2.2.3 High-Level Petri Nets

CPNs are an expressive language allowing the modeling of states, events, parallel, conditions and recursive routing. However, they are not appropriate for modeling real-life applications. According to ISO/IEC 15909 [35], the problem with them is the explosion of the number of elements of their graphical form when they are used to describe complex systems. High-Level Petri Nets (HLPNs) distance themselves from CPNs by extending the CPNs with high-level concepts to overcome CPNs limitations. The first meaningful extension was the Predicate/Transition Nets by Hartmann Genrich and Kurt Lautenbach in 1979 when they presented their first paper [28]. In these nets, the tokens were given values enabling the distinguishing of different types of tokens, which are usually illustrated with different colors. This, in turn, enables the transitions to have different behaviors for a given type of tokens. In CPN, the same behavior of the net could be produced. However, the complexity of the CPN increases at a much higher rate as more kinds of tokens are inserted. An illustration of a CPN and an equivalent P/Tr-net (Predicate/Transition net) can be found in figure 2.2.
2.2. WORKFLOW MODELING LANGUAGES

Other popular extensions include extending the CPN with temporal behaviour such as timeouts and hierarchy, allowing subnets within one net.

2.2.4 Workflow nets

In a paper presented by van der Aalst[62], a proposal for mapping the control-flow aspects of workflows onto Petri Nets is described. In that paper, they model tasks as transitions, conditions as states and cases as tokens. They address these types of PNs as WF-nets(WorkFlow nets). Their example of a model for processing complaints can be found in figure 2.3. A WF-net satisfies 2 conditions:

- It has one, and only one, input place \( i \) and output place \( o \). A token in \( i \) corresponds to

![Figure 2.2: Classic Petri Net and Colored Petri Net Comparison](image)
Figure 2.3: Process modeled in a workflow net

a case to be handle and a token in $o$ corresponds to a case that has been handled.

- It has no dangling tasks and/or conditions.

2.2.5 YAWL Language

Although PNs are expressive, their expressiveness is hindered by the following limitations.

- PNs do not directly support the execution of multiple instances of subprocesses. These can be implemented, but the user is forced to split and synchronize the instances explicitly.

- The branching mechanisms of PNs are limited to ANDs and XORs. This means that users are limited to parallel execution (AND) or choice (XOR), as there is no operator that allows the execution of only some branches (OR).

- There is no way to model cancelation of tasks that affect non-local parts of the process.

YAWL [33] is a language based on PNs that solves these limitations. By adding some new operators, it explicitly enables (1) the specification of advanced branching where any number of branches can be selected for execution, (2) the modeling of multiple instances where users can specify the upper and lower bound for the number of created instances and the synchronization method for those instances (whether an instance has to wait for other instances to finish to proceed) and (3) the specification of cancelation logic where the user can connect the removal of a token from one place to the removal of tokens in other places.
2.3. COMPARISON BETWEEN MODELING LANGUAGES

2.2.6 UML Activity Diagrams

UML activity diagrams are a special case of the UML state diagrams that can be used to model workflows. Yet, these are seldom referred to as a standard method for modeling workflows [68].

The fundamental unit of executable functionality element within a UML 2.0 activity diagram is the action [54]. An action represents a single step within an activity, which is equivalent to a task in a workflow[40]. It takes a set of inputs and produces a set of outputs, though they may be empty sets. These actions can begin their execution when their input conditions are satisfied.

The control-flow aspects of the activities are conditioned by control nodes. These include the decision nodes for conditional routing and synchronization bars for parallel routing. Finally, these activity nodes are connected with activity edges.

2.2.7 BPEL4WS

A graphical representation of BPEL4WS looks like a flow-chart, where elements are activities, which can be basic or structured. The set of basic activities permits the invocation of Web Services (invoke), the provision of Web Services (receive), variable updates (assign), fault signing (throw), timed pauses (wait), and empty activities (empty). Structured activities allow the definition of complex control-flow mechanics, namely sequential (sequence), conditional (switch), parallel (flow) and recursive routing (while). Moreover, the pick activity allows the process to wait for one of a set of specified events and perform the associated action for that event. Finally, using the link tag allows the specification of the ordering between activities inside a parallel routing in a DAG (direct acyclic graph) manner.

2.3 Comparison Between Modeling Languages

2.3.1 Expressiveness

The comparison of the workflow patterns each mentioned language supports is presented in table 2.1. In terms of expressiveness, YAWL is the best choice, since it supports every pattern with the exception of the Implicit Termination. This implies that the user needs to guarantee that the final node is reached, or else the process will not be concluded even if there are no more tasks to execute. Actually, this limitation was added to make the designer to think about the finishing conditions of his workflow [33]. After YAWL, the language that supports the highest number of patterns is UML with 16, then both HLPN and BPEL4WS
with 13 and partially supporting 1 (see table 2.1). However, it is important to notice that the sheer number of supported patterns does not imply that one language is more appropriate to model workflows than another. Depending on the domain, some patterns might appear less frequently. Another factor is that some of the workflow patterns that are unsupported directly might be easily implemented, while others patterns might be extremely complex and sometimes unfeasible.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>HLPN</th>
<th>YAWL</th>
<th>UML</th>
<th>BPEL4WS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Sequence</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>2-Parallel Split</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3-Synchronization</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>4-Exclusive Choice</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>5-Simple Merge</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>6-Multi Choice</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>7-Synchronizing Merge</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>8-Multi Merge</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>9-Discriminator</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>10-Arbitrary Cycles</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>11-Implicit Termination</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>12-MI without Synchronization</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>13-MI with a Priori Design Time Knowledge</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>14-MI with a Priori Runtime Knowledge</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>15-MI without a Priori Runtime Knowledge</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16-Deferred Choice</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>17-Interleaved Parallel Routing</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+ / -</td>
</tr>
<tr>
<td>18-Milestone</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>19-Cancel Activity</td>
<td>+ / -</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>20-Cancel Case</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison between HLPN, YAWL, UML and BPEL4WS [33] [68] [69]

2.3.2 Analyzability

A strong reason for the use WF-nets/PNs is the fact that they are analyzable. PNs can be analyzed for properties such as soundness and measures including response times, waiting times, and throughput [61]. The soundness properties ensure process termination and the absence of dangling references, deadlocks, and livelocks [9]. A WF-net is sound[62] if it has the following properties:

- From any reachable state $M$ from state $i$, there is a sequence that leads $M$ to $o$. This guarantees that all cases can terminate.
2.3. COMPARISON BETWEEN MODELING LANGUAGES

- There is only one state reachable from $i$ with at least one token in $o$. This prohibits the existence of dangling states/transitions.

- There are no dead transitions. Which means that all the transitions can be enabled.

This verification can be made using tools such as Wolfan [63], and most PN editors have this feature.

YAWL can also be analyzed for soundness. Although the semantics of YAWL is not given in terms of Petri nets, the same analysis algorithms can be used. The author proves formally that many results from Petri Nets can be carried over to YAWL [55].

The analysis of UML activity diagrams has not been explored as much as PNs. Very recently, a paper was published showing some work done in the analysis of UML activity diagrams [47].

Finally, the correctness of BPEL4WS can only be verified at runtime [13]. Because of this, the work in verifying BPEL4WS has been focused in capturing the formal semantics of BPEL4WS in terms of formalisms such as PNs, process algebra, and finite state machines [42].

2.3.3 Usability

The PN-based modeling languages are the simplest to use. Complex behavior such as parallel routing and loops are all modeled in terms of a small set of constructs (places, transitions, arcs, and tokens). Moreover, the language is mathematically formalized, despite having a graphical nature, leaving no room for ambiguous interpretations.

The YAWL language is very similar to PNs, except that it introduces some new constructs. This means that YAWL shares the same set of advantages with PNs. The new constructs it uses, such as the token removal operator used for cancelation patterns (19,20), might confuse some users. However, they do not influence in any way the basic constructs that are found in PNs, so the user has the choice of leaving those constructs untouched.

UML activity diagrams have some similarities with PNs, in the sense that they are graphical. Yet, they are not formalized by OMG[54]. The main argument UML activity diagrams have against PNs is the fact that UML activity diagrams are reactive, which is better suited to deal with workflows than PNs that are active. In activity diagrams, transitions are triggered in response to events following the event-condition-action rule. These events can be the termination of an activity, or some input from the user. In PNs, on the other hand, tasks are only enabled when it’s execution conditions are fulfilled, and the actual execution of the task can be postponed until later. Because of this, PNs are considered active.
BPEL4WS has the problem of being language based, making it harder to learn than graph-based languages. The language requires the designer to become familiar with numerous concepts before being able to create processes efficiently. Many BPEL vendors offer visual interfaces that create a graphical representation of the process by representing structural activities as nodes, transitions and links. This facilitates designer’s job to some degree, but it still cannot compare with the usability of graph-based languages. Furthermore, its original design goals say clearly that BPEL4WS is not concerned with graphical representations or a graphical design methodology [44].

2.3.4 Support

PNs have been subject to studies many years before the term workflow management existed. Many applications exist today that support PNs, including graphical interfaces and verification tools.

UMLs activity diagrams, on the other hand, have only been considered to model workflows recently. Thus, there are few applications that support UML activity diagrams for workflows.

YAWL has only one implementation that can execute the instances of YAWL processes. Fortunately it is quite complete, having a visual editor and a web service broker, in addition to the engine.

Although BPEL4WS has not been around for as long as PNs, due to its popularity, there is a large variety of solutions (both proprietary and open-source) for BPEL4WS on the market, which comprise not only of a wide variety BPEL engines, but also tools for verification of BPEL4WS processes and visual editors.

2.3.5 Summary

PNs are a good way to model workflows, since they are simple, formal, and graphic. Moreover, they have an abundant array of analysis techniques to prove properties and performance measurements.

YAWL is based on PNs, inheriting most of their characteristics, and achieves an unprecedented expressiveness level. The only drawback of YAWL is the fact that it only has one implementation available at the moment.

UML activity diagrams are similar to PNs and have the advantage of being reactive. However they are not formalized and their analyzability is still in an immature state.
BPEL4WS has many providers and is a widely accepted standard for modeling business processes [53]. However, the language has a steep learning curve, requiring some effort and time to become acquainted to the language.

In conclusion, YAWL seems the most suitable choice as it combines expressiveness, analyzability, and usability.

2.4 Workflow Engines

All workflows need an execution environment that allows the execution of workflows and gives the user control over their execution. In this section we discuss the various workflow engines that are available on the market today.

Most of the engines described below are actually full workflow management systems since workflow engines are rarely implemented on their own. Because of this, this section will also be an overview of existing workflow management systems.

2.4.1 DAGMan

DAGMan [57] is a service for executing multiple jobs with dependencies in a declarative form. It is well integrated with Condor, to whom it submits its jobs. Condor’s role is to find resources for execution those jobs, and, because of this, it is considered a grid workflow system. Users create workflows in the form of acyclic graphs where jobs are defined as vertices and dependencies are arcs. This graph is specified with a description file that is submitted as input to DAGMan.

DAGMan divides jobs in two categories: (1) jobs to be submitted to Condor and (2) jobs to be submitted to Stork [41], a data placement server. The former defines the main units of work of the DAG while the latter is used to transfer the data to the machines that will be running the jobs that are submitted to Condor. By using Stork, data transfer jobs are carried out in a schedule and fault-tolerant fashion. As far as jobs are concerned, DAGMan gives the user some control over the control-flow of the DAG. Users can specify the priority level of each job to define the order of the jobs to be submitted in cases where more than one job can be submitted, set the number of times retries until a job is rendered as failed, as well as the max number of instances for each job, and abort the process entire process. Users can also define scripts to run before or after a job, which is useful when the input and output data need to be processed. The dependencies between the jobs are modeled in a parent - child manner where parent jobs must be dispatched before the child jobs. DAGMan also allows users to program recursively, which means that a DAG can be used in other DAGs.
After the DAG is running, DAGMan allows the monitoring of the through a log file where users can see the progress of their DAG and allow their removal. Also, when a job in a DAG fails on its own accord and produces an error result, the process continues until further progress is not possible without the failed jobs. This allows the user to repair the failed nodes and resubmit the DAG without re-executing the completed jobs.

As a final note, by using dot [6], a language for drawing directed graphs, users can generate a graphical visualization of the DAG.

2.4.2 P-grade

P-grade [46] is a workflow management system for the development of complex grid applications supporting the design, execution, monitoring and performance visualization. It supports the execution of applications on different environments, such as Condor, Condor-G, Globus, and Dagman.

In P-grade, a graphical editor is used to design workflows. They use their own workflow language and the language’s main elements are, jobs, and connectors. Jobs properties include the job type (sequence or in parallel), the path to the job executable from the local file system and the grid environment where it is executed. A job also has ports, which symbolize the input and output files of the job. Connectors are used to connect the input ports and output ports together, defining how the files are passed between jobs and how the workflow evolves. Workflows are then submitted to the system and executed. The graphical editor is also used to monitor workflows.

2.4.3 BPEL4WS

BPEL4WS is one of the key standards accelerating the adoption of SOA (service oriented architecture) [12]. Promoting the integration and assembling of systems from individual Web Services, it grants organizations a new level of agility enabling them to quickly adapt their processes as needed.

In BPEL4WS users create processes that specify the way that existing services are chained together as a workflow. That specification takes the form of an XML file described by the BPEL4WS specification [4]. However, the BPEL4WS process description file generally does not stand alone. It usually refers to a WSDL (Web Service Description Language) [15] describing the service’s interface. To be able to execute the process users need a BPEL4WS engine, such as ActiveBPEL [2] that processes the specification and deploys it as another Web Service.
2.5. COMPARISON BETWEEN WORKFLOW ENGINES

2.4.4 YAWL

YAWL [3] is the only workflow management system that supports the YAWL language at the moment. YAWL is composed by many loosely coupled components that interact with each other through web services. The modeling of workflows specifications is done using the YAWL designer and the specification is submitted to the YAWL engine. The YAWL engine stores the specification at the YAWL repository, after which the instances of those workflows can be executed. Lastly, the interaction between the YAWL engine and its environment is mediated by three services. These services allow the invocation of web services, the handling of worklists for users and the coordination with other workflow systems. Additionally, more services can be added as custom services.

2.5 Comparison Between Workflow Engines

2.5.1 Expressiveness

The YAWL engine and the BPEL4WS engine use the YAWL language and the BPEL4WS language, respectively. The expressiveness of these languages has been described in section 2.2.

Like all languages based on DAGs, DAGMan’s and P-grade do not allow the modeling of cycles. Furthermore, they also have no operator for specifying conditional routing.

2.5.2 Environment

Both BPEL4WS and YAWL communicate with the environment through Web Services. In the case of YAWL, the engine described in the initial implementation [3] only had the worklist services implemented. Currently, YAWL supports Web Services but has some limitations [25]. The service invoker does not support sending call back addresses to asynchronous services and it cannot invoke web services with complex data types as their parameters.

In DAGMan, every job is executed through Condor, which means that it cannot directly use jobs in other environments. As the environment for the execution of NLP tasks is not restricted to Condor, it would drastically decrease the value of the engine.

Presently, P-grade does not support Web Services present in BPEL4WS and YAWL, but supports a large variety of grid technologies, which compensates for that limitation.
2.5.3 Summary

The limitation of DAGMan in the Condor environment puts it in disadvantage against BPEL4WS and YAWL that support Web Services allowing them to interact with a multitude of environments. However, the main problem is the fact that mapping from YAWL language to the DAGMan’s language would not be trivial, or even impossible, due to the fact that the DAGMan’s language semantics is not nearly as expressive as YAWL’s. P-grade also shares these problems, although it provides support for a wider number of grid environments.

BPEL4WS is a good candidate for the workflow engine because of the large array of suppliers of BPEL4WS engines (Active BPEL, Apache ODE, Oracle BPEL process manager, Sun ESB). Furthermore BPEL was approved as a standard [53] by OASIS \(^1\). However, there is currently no work done in mapping the BPEL4WS language to YAWL, although there is work done on mapping YAWL into BPEL4WS [5]. This is mainly because BPEL4WS is not as expressive as YAWL in terms of support for the workflow patterns.

YAWL is another good choice for the workflow engine mainly because the mapping would be much easier. The drawback with YAWL is the fact that the only implementation of YAWL lacks support for some of the Web Service functionalities described above.

In conclusion, there are two viable alternatives for the workflow engine. We can use a BPEL engine, trying to map the YAWL language to BPEL4WS, and model the unsupported patterns in BPEL4WS by either finding an equivalent combination of structured activities, or creating Web Services that simulate these patterns in runtime. Still, we will choose the YAWL engine, since it allows us to avoid the problems of mapping from one language to another.

\(^1\)An international open standards consortium.
Before describing the architecture of our system, we need to understand how the YAWL engine works. This chapter will cover the elements of YAWL’s architecture that are relevant for this work.

### 3.1 YAWL engine

The core component in YAWL is its engine, which is responsible for deploying, validating, and executing workflow specifications. After a workflow specification is deployed, instances/cases of that workflow can be launched. A key aspect that distinguishes the YAWL from other workflow management systems is that, while in traditional workflow management systems, the engine is responsible for taking care of “What”, “When”, “How” and “By whom”, the YAWL engine only takes care of “What” and “When” [3]. It does this by keeping a list of available services, called Custom Services, and their respective URLs where the service is deployed in its repository, and then the binding for each node to each service is defined, by the user, within the specification. Having these two elements, when the engine determines that a node is to be executed, the engine fetches the URL of the service to invoke for that node and sends it an HTTP message to the service, announcing that the node is waiting to be executed. The HTTP interface between the engine and the service is defined using a REST architecture [23].

### 3.2 Custom Services

The YAWL distributions include a handful of services, including a service to invoke Web Services, but users are free to create additional services (Custom Services) that can be invoked during the execution of workflows. To do this, the following must be done:

1. The service must be implemented and deployed. The service can be implemented in any language or platform but must implement the same interface of the engine, meaning that it must be able to parse the HTTP messages sent by YAWL during the execution of the workflow, and respond accordingly. In the situation that we are implementing
our service as a Java servlet, YAWL provides Java classes, such as the “InterfaceBWeb-
sideController”, that provide the necessary abstractions that allow the service to be
implemented more efficiently.

2 After the service is deployed, the service needs to be registered in the YAWL engine.
This can be done using the “interface B” of the engine which we will cover below.

For applications to communicate with the engine in order to register new custom services, de-
ploy specifications, and launch cases, the YAWL engine provides various interfaces depending
on the request type. For instance, to deploy workflows and launch cases we would interact with
the “Interface B”, while requesting case information would require us to exchange messages
with the “Interface A”. Once again, if we implement our applications in Java, YAWL provides
classes (“InterfaceA_EnvironmentBasedClient” and “InterfaceB_EnvironmentBasedClient”) that provide methods that converts the HTTP messages to Java objects and vice-versa.
YAWL provides some applications that use these interfaces, one of them is the YAWL editor
which is a Java applet used to design specifications graphically, and the other is the Resource
Service, which is a browser based application that allows the deployment of specifications
but more importantly, the management and assignment of tasks (nodes that are associated
with tasks that are allocated to a person or group). More details will be discussed about this
service in section 3.4.

With this done, whenever a node that references that service is executed, the YAWL engine
will send it a message with identification of the execution’s instance. The service must check-
out the Work Item, where the input parameters of the node’s execution are stored. Checking
out the Work Item also guarantees that it will not be checked out again by another service,
so that a Work Item will not be executed more than once. After executing the service has
completed its work, it must check the Work Item back in passing its output parameters. The
checking out and back in of the Work Items are also done using the “Interface B” of the
YAWL engine.

3.3 YAWL specification

The YAWL specification is the document that describes the process to be executed by the
engine. By parsing this document, the YAWL engine builds the internal representation of the
workflow and deploys it, allowing instances to be launched from that point on. We will be
giving an overview of the main elements within a YAWL specification, as it will be needed to
understand some aspects of our architecture.

A YAWL specification is an XML document enclosed by the root node “SpecificationSet”,
which has one or more “Specification” elements, each representing one particular workflow
3.3. YAWL SPECIFICATION

specification. This means that more than one workflow specification can be uploaded to the engine using the same specification file, and upon uploading the specification the engine deploys each specification separately. Another element that can be placed as child of the “SpecificationSet” element, but is irrelevant for the engine, is the “layout” element, which contains graphical information that is only used by the YAWL editor, such as the positions of the nodes and edges. Figure 3.1 provides an example of a YAWL specification with these elements. The “Specification” element describes the workflow from three perspectives [3]:

1 Control-Flow perspective - This perspective describes in which order are the nodes executed within the workflow.

2 Data perspective - This perspective describes how is the data updated during the execution of the workflow.

3 Operational perspective - This perspective describes which Custom Service is associated with each node.

A “specification” element, shown in figure 3.2, has a attribute named “uri”, which is a unique identifier of the workflow, and two workflows with the same identifier and version cannot be deployed simultaneously. The specification’s child elements consists of a “metaData” element, which provides information about the workflow (version, creator, documentation, etc...), and a group of “decomposition” elements. A decomposition has a “type”, which can either be “WebServiceGatewayFactsType”, which associates the decomposition with YAWL service, or a “NetFactsType” which associates the decomposition with a series of coordinated tasks, where each task is associated with a decomposition. A simple workflow would consist of a single “NetFactsType” decomposition that describes how tasks are coordinated and various “WebServiceGatewayFactsType” decompositions describing what to do with in each task. However, a task within the “NetFactsType” can be decomposed into another “NetFactsType” decomposition allowing an hierarchical structuring of workflows. In both types, the decomposition needs to declare it’s input and output parameters, as well as local variables in the case of the “NetFactsType” decompositions. Every workflow specification needs one and only
CHAPTER 3. YAWL ARCHITECTURE

Figure 3.2: Specification Element Example

one of the decompositions that has the “isRootNet” property set as “true”, which is the decomposition that will be executed when the workflow instance starts.

3.3.1 “WebServiceGatewayFactsType” decomposition

The decompositions of the “WebServiceGatewayFactsType” type require the user to specify the YAWL service to associate with that decomposition, which is done by appending an “yawlService” element. We can see an example in figure 3.3, where an Web Service is called. Aside from the service’s input and output parameters (“isbn” and “GetBNQuoteResult”), additional parameters must be declared for the YAWL service (“yawlWSInvoker”). In this case, the parameters used to determine the WSDL location (“YawlWSInvokerWSDLLocation”), the port (“YawlWSInvokerPortName”), and the operation to invoke (“YawlWSInvokerOperationName”). The arguments to be passed varies depends exclusively on the YAWL service, since the engine is oblivious to which parameters need to be passed to each Custom service. The Custom Service itself gets the values of these parameters and decides what to do with them. In the case of the “yawlWSInvoker” in the example, the service will use the WSDL location, port and operation to invoke the Web Service and uses the rest of the parameters as the arguments it, in this case the “isbn”. The same happens with the output parameters, where the services must pass an output parameter with the name “GetBNQuoteResult” when it checks the Work Item back into the engine and parameters with other names are ignored since they are not declared in the decomposition.

3.3.2 “NetFactsType” decomposition

“NetFactsType” type decompositions (figure 3.4) are the definition of the actual workflow. The control-flow aspects of the decomposition are described in the “processControlElements” element. Within this element, there is an “inputCondition” element and an “outputCondition”, which can be associated with the input place and output place described in Workflow nets (Chapter 2.2.4, which refer to the start and end of the net, and an arbitrary number
3.3. YAWL SPECIFICATION

3.3.2.1 “Task” element

“Task” elements (figure 3.5) also need an “join” and “split” elements which can be either “and”, “or” or “xor”. Concerning the “split” element, an “and” means that all the edges in the “flowsInto” are activated upon the termination of the task, while a “xor” split only activates one of them. To do this, the user has to declare an extra “predicate” element in the “flowsInto” element. The “predicate” element has also one property called priority which allows to specify the order in which the predicates are tested. The “or” split is similar to the “xor” split except in that more than one edge can be activated. The different types of join are similar to the splits. The “and” join means that when every edge that connects to the task is activated the task can start, while the “xor” join requires only one activated edge and the “or” join needs an arbitrary number of them.

The query for the predicate is written using the XPath language [36], which allows easy access to the in the workflow variables since YAWL presents them in a XML format. For instance, if we had a net with the id set to “xpto” with variables named “x” and “y” and values “10” and “20”, respectively, the YAWL engine would present it as “<xpto><x>10</x><y>20</y></xpto>”, so to access the value of the variable “x” we
Figure 3.4: “NetFactsType” Decomposition Example
3.3. YAWL SPECIFICATION

would have to write 
\( \text{/xpto/x/text()} \). The text “\text{xpto/x/}” part describes which element in the XML document that we want to reference, which is “x”, and the ”text()” function is used to retrieve the value inside that element, which would be “10”.

Tasks also need to be associated to another decomposition, which is done by specifying the decomposition in the “decomposesTo” element. However, it also needs the specification of the values to be passed as the input arguments of the decomposition and what to do with it’s output arguments. The “startingMappings” element is used to accomplish the former, and each ”mapping” inside it will define a input parameter, while the “completedMappings” allow you to assign the output parameters of the node to variables of the net. The values that are passed to the parameters are defined in the “expression” tag using the attribute “query”. These queries are defined by using XPath expressions as well. In figure 3.5, we can see that the starting and completed mappings reference the input and output arguments in the “WebServiceGatewayFactsType” decomposition (figure 3.3).
3.4 Resourcing Service

One of the most elaborate Custom Service that comes with the YAWL package is the Resourcing Service [32], which is accessible through a web interface. This Service allows the assignment of Work Items within the workflow to resources, which can be human or non-human, but we will refer to these as users since it is the only resource that is available in the service presently.

3.4.1 User Model

The Resourcing service has its own entity model for users in which each user has a set of roles and capabilities. Users can also be part of an organization, which can, in turn, be part of another organization. Inside an organization, the each user also has a position.

The role is basically what the user does and based on it a workflow specification can, for each task, define a set of rules on which the allocation of each task to a certain user is decided. For instance, if we would have a task “repair car” we would want to attribute it to the role “mechanic”. Furthermore, a role can be part of another role, in which case the user that has a certain role will have all the roles that belong to that role. Capabilities on the other hand are more specific to certain users, like some user can only repair cars of a certain model. Finally, organizations can define positions and allocate them to the users.

Users also have 3 task queues: offered, allocated and started queues. When a task associated with this service is announced by the YAWL engine, this service puts the task in the offered queue in every user that is eligible. Offered jobs can then be allocated to users, and the ability of a user to allocate a job to another user depends on his position in the organization, roles and capabilities. This can be done by directly selecting the user, or specifying a role or position and let the system allocate it automatically. When a job is allocated to a user, it is moved to the allocated queue of that user. The user is then able to start the task, and, by doing so, will make the system checkout the respective Work Item from the YAWL engine. After starting the task, the user must fill the output parameters of the Work Item and submit it. Doing so will make the system check the Work Item back to the engine allowing it to proceed to the following tasks.

3.4.2 Resoucing Specification

Although the Resource service is a Custom Service, the specification of these nodes differs slightly from other services, in the sense that an extra “resourcing” element must be placed within the “task” element. This element must have three children: “offer”, “allocate”, and
3.4. RESOURCING SERVICE

Figure 3.6: Resoucing Element Example

"start" node. These define the rules by which the tasks are offered, allocated and started, respectively.

The "initiator" can be set to "user" or "system", depending on whether the attribution of the task is left to a user or to the system. "Offer" elements that are delegated to the system must have a distribution set, declared by the tag "distributionSet", which declares the initial set of users that the job is offered. To allocate the job automatically the "allocate" tag must have a "allocator" defining the algorithm through which the user is chosen. Some examples of possible algorithms include the random algorithm, which chooses a random user from the users that are offered the job, and the round-robin algorithm that distributes the tasks evenly between the users in case the task is called multiple times. In contrast, the "start" does not need additional child elements, since it only decides whether the job is started automatically or not. An optional element that can be added to the "resourcing" element is the "privileges" element which allows the specification of the privileges given to the users, which include, whether the task can be skipped, delegated or suspended.

Figure 3.6 provides an example of the use of the "resourcing" element. If this where used in a workflow, the task associated with the node would be offered and allocated to the user with the id "PA-64a1b8ed-76e8-4d64-8a11-ad58ea32eaf0", which is the id that is generated by the system when a new user is created. Starting the task on the other hand is done by the user, since the "initiator" is set to "user" in the "start" element.
3.5 Summary

This chapter described the elements of the YAWL architecture that were relevant for our system. The main component in the YAWL’s architecture is its engine, which is responsible for managing the task execution order in the workflow. The actual work is done by the various Custom Services that the engine invokes during the execution of the workflow, and YAWL provides an interface allowing more services to be implemented and added to the system.

The workflows are defined and published in the form of an XML file, where the elements of the workflow are defined. These include the tasks to be executed within the workflow (Operational perspective), the variables where the data is stored (Data perspective) and the dependencies between the tasks (Control-Flow perspective).

The Resourcing Service is one of the Custom Services that comes in the YAWL package, which allows the allocation of Work Items to resources, such as users. This allows them to partake in the execution of the designated nodes within a workflow.
In this chapter, we provide an overview of the architecture of our system, which is illustrated in figure 4.1. Our system is composed by a number of loosely coupled components that are connected by either HTTP [65] or RPC [11]. The advantage of this is the fact that the different components can be deployed in a distributed manner and each of the components can be developed and upgraded independently. It also allows the insertion of additional components in the future easier. The architecture is composed by 2 web containers, one where we will deploy the YAWL engine and another for our own workflow server. On the YAWL engine’s side, our work will be focused on extending the YAWL’s operational perspective, by implementing additional Custom Services. The following sub-sections describe the each of these components in detail.

4.1 Custom Services

As mentioned in section 3.1, the YAWL engine provides an extensible interface that allows the implementation of additional services that are executable during the execution of workflows, referred to as Custom Services. These define what type of tasks the workflow can perform during the execution of each node.

The services are deployed in the same web container the YAWL engine is deployed so that the messages between the engine that service can be exchanged faster. The Custom Services that were implemented are based on the tools commonly used in NLP applications and environments, such as the Condor Service and also the requirements for some features of our system, such as the EventLogger Service. The following list presents the services that were added:

1. Condor Service - This service provides the interface to interact with Condor. This is needed since many tasks in scientific workflows require vast amounts of CPU.

2. EventLogger Service - This task allows the monitoring of workflows. This will be explained in detail in section 5.3
CHAPTER 4. SYSTEM ARCHITECTURE

Figure 4.1: Architecture Overview

Figure 4.2: Server Side Architecture
4.2 Workflow Model

Before we start describing the Workflow Server and Interface, we will show how our workflows are modeled within the system. This model is based on YAWL specifications so that the conversion to the YAWL specification is easier. Figure 4.3 presents a simplified version of our workflow model. Each workflow is represented by a “Workflow” entity. Workflows are identified, not only by their name, but also by their namespace, to avoid name conflicts. It also has the version of the workflow which identifies the version of the workflow, which is used for workflow versioning described in section 5.2.3. It also has a creator and documentation.

Nets define composite executable elements within the workflow. For a workflow to be executable, it needs at least one of these nets. Furthermore, it needs declare which one of the nets is to be executed when the workflow is executed. The remaining nets can be executed during the execution of the root net.

A net has a set of variables modeled by the “Variable” entity. Variables are the containers of the data that is passed between nets and nodes. There are 3 types of variables: input, output and local variables. Input variables contain values that are passed as parameters when the net is executed. Upon the end of the net’s execution its results are returned in the output variables. Local variables are sorely used within the net. In the particular case of the root net, its input and output variables define the input and output parameters of the workflow. Variables are referenced by their names which are given by the user like most programming
languages and there is no name conflict when two variables have the same name in different
nets and workflows. Finally, variables need to have a data type. At present, only a preset
number of primitive types, such as strings and integers, can be used. This limitation imposes
some restrictions on the services that can be invoked. Some services, such as the Condor
service and HTTP service, do not suffer from this limitation since all their parameters are
strings, but in the case of the Web Service Invoker, it will not be able to invoke Web Services
that use complex types as arguments. The reason for this is that the Web Service Invoker
that YAWL provides would have to be re-implemented since the service itself only supports
4 primitive types.

Aside from variables, a net also has a set of nodes. A net needs to have one “Start Node”,
which is where the net starts the execution, and a “End Node”, where the net ends. These are
created automatically when a net is created. Aside from those, a net can have an arbitrary
number of “Executable Node” entities. Executable nodes define atomic units of work.

There are two sets of information that are associated with nodes. The first set contains
information regarding the relationship between the node and the rest of the net. The node
will have a list of ingoing and outgoing edges, represented by the “Edge” entity, and the join
and split type. These represent the conditions on which the node is allowed to execute as
well as state of the net after it is finished executing. The second set is related to the actual
work that is to be executed in the node. This set is composed by the input and output
arguments, represented by the “Argument” entity, of the node and the respective mappings
for each of those arguments, represented by the “Input Mapping” and “Output Mapping”
entities, respectively. Arguments are defined by their name and type, and are only used to
help the user specify the mappings. Each input mapping puts the value of a query, which
can use the variables of the net to an input argument of the node and each output mapping
maps the value of a query that can include any output argument of the node to a variable of
the net.

As can be seen in figure 4.4, in the category of executable nodes, there are various node types.
These can be divided into two groups: atomic nodes and composite nodes. The difference is
that composite nodes can be further decomposed recursively until they become nodes of the
atomic nodes, which cannot be divided further.

There are only three atomic node types. The first is the YAWL Service Node, which, as the
name implies, will execute a Custom Service, specified by its identifier. In these nodes, the
arguments of the node are defined upon the creation of the nodes within the workflow. The
second one is the Resourcing Node, which uses the YAWL’s Resource Service for interacting
with users, that has a distinct interface. The last one is the Sub Net Node, associated with
the execution of another net within the same workflow. In this case, the arguments of the
node are the input and output arguments of the net that is to be executed by the node.

Composite node types, exist in a larger variety and include the Condor Node, the Web Service Invoker Node, and the Subnet Node. These are expanded into combinations of the above nodes, before being submitted to the YAWL engine by the Marshaller component, described in 4.3.2. For instance, the Web Service Invoker Node is expanded into a Subnet node and referencing a net that uses the YAWL Service Net which calls the WSServiceInvoker.
4.3 Workflow Server

An architecture only involving the Custom Services described in section 4.1 would suffice, since users could write their workflows using the YAWL editor, submit them to YAWL engine, and execute it afterwards. However, we decided to create our own server since there are some issues with this architecture:

1. As referred in section 3.3, YAWL is oblivious of the arguments that each Custom Service needs to work properly, which means that in order to use these services, the user has to either consult the developer of the service or read the documentation of the service. This makes the process of designing workflows more error prone and harder to learn. As mentioned in section 1.2 one of the goals in our system is to make the process of workflow specification simpler.

2. The YAWL language does not allow one specification to reference another specification directly. This can be done by copying every decomposition from the referenced specification, which, once again, makes developing workflows more error prone since the specifications can grow larger.

3. This architecture would not have any way to monitor the progress of the workflow instances nor detect or recover from errors. These features are essential in large workflows or workflows with tasks that last for hours or days.

4. Some services are not convenient to be invoked as they are. Take for instance the Condor interface. To submit a job, the user would have to choose a cluster (or create one), send the input file and start the job. Afterwards, he would have to periodically check whether the job has been completed. After completed, he would need to retrieve the file where the job’s output had been written. This would involve creating at least 3 nodes, one to submit, one to check for completion, and one to retrieve the outputs and also some control-flow logic such as a loop until the job is completed. We could make this more complex by creating another node that starts if the job has failed and restart it upon user input and so on. The system should be able to hide such complexity from the user by presenting a single node.

In order to cope with these problems, we added a additional server, called Workflow Server. This server acts as a central server, where entities within the system, such as users, tasks, and workflows are managed. Its main task is to receive the workflow description from the user, created using our interface (described in section 4.4), and execute it. It does so by producing a YAWL specification file, leaving the actual execution of the workflow to the YAWL engine. From the specification submitted by the user to the server until the server
produces the YAWL specification, some changes are made to the structure of the workflow, such as inserting additional tasks to keep track of the progress of the workflow and adding error recovery tasks (the way this is done will be illustrated in chapter 5.3).

The architecture of the server is displayed in figure 4.2 and the rest of this section will describe each of it’s components.

### 4.3.1 Database Gateway Layer

The workflows, users and other entities in the system are stored in a database. The access to these objects is abstracted into a Database Gateway Layer, which are a set of components that provide the CRUD \(^1\) functions for them. The current implementation of this layer stores the objects in a MySQL [49] database, using the Hibernate 3 [31] framework deal with of Java object persistence. Furthermore, this implementation possesses unit testing classes, using JUnit 4.5 [10], to test whether these classes are functioning as desired. These tests are run in an alternate database.

The following list presents the different Components in the Database Gateway Layer in the system:

1. **Workflow Gateway** - This component is used to access the entities related to workflows in the database. The Workflow Gateway deals with the entities related to workflows, as well as their instantiations.

2. **Organization Gateway** - The users that are registered in the system are stored using this component.

3. **Data Type Gateway** - The information about existing data types accessed using this component.

### 4.3.2 Marshaller

To execute workflows in the YAWL engine we need to convert our workflow objects into YAWL specifications. This conversion is attributed to the Marshaller component. This component is responsible for converting workflows into YAWL specifications. Furthermore, this component is also responsible for marshalling and unmarshalling workflows, so that they can be stored in the database as XML strings.

The Marshaller is defined by several components that are listed below:

\(^1\)Create, Read, Update, and Delete
CHAPTER 4. SYSTEM ARCHITECTURE

1 Workflow Marshler and Unmarshaller - Converts the textual specification of the system workflows to domain objects and vice-versa.

2 Workflow Pre-processor - Preprocesses workflows before converting and sending them to the YAWL engine. Some tasks that are done in this component include the addition of nodes to log important events in the workflow, as explained in section 5.3, and expanding composite nodes into atomic nodes.

3 YAWL Converter - Converts a system workflow into a YAWL specification string.

4.3.3 YAWL Interface Client

The YAWL Interface Client is the component that deals with the communication with the YAWL engine and the Resource Service. As it was mentioned earlier, the YAWL package provides a set of classes that provide the methods for communicating with the engine. This is useful, because it spares us the work of building the messages to send, as well as the parsing of the received messages. Thus, this component will be composed by the interface clients provided by YAWL. The following is a list of the interface clients in YAWL:

1 Interface A Client - This is the client that deals with the general operations of the YAWL server. This interface is used to register new YAWL Services and to uploads specifications.

2 Interface B Client - This is the client that is more workflow oriented. It allows the execution of workflow instances and management of Work Items.

3 Resource Client - This is the client that communicates with the Resource Service. This allows the management of user work queues and provides data regarding users and organizations within the Resourcing Service.

4 Workqueue Client - This client contains the functionality of the Interface A Client and Interface B Client as well as some other additional operations.

4.3.4 Service Callback Interface

In some Custom Services, they might need to report back to the Workflow Server. This is done using the Service Callback Interface, which is a set of services to access the particular functionalities of the server. A present, the only Custom Service that uses this interface is the EventLogger Service because it must report the event back to the Workflow Server so that the event can be registered.
4.3. WORKFLOW SERVER

4.3.5 Manager

The mediation between the previous components within the system is done in the Manager Layer, which composes the business logic of our system. This layer is divided into several components:

1. Workflow Manager - This component is used for operations that manipulate workflows. One of the jobs of the Workflow Manager is to guarantee that the model in our server is synchronized with the YAWL engine. For instance, when we launch an instance of a workflow using the Workflow Manager, it will use the YAWL engine’s interface to launch an instance of the workflow but also create an object in our server that represents that instance, which references the YAWL engine’s workflow instance by its id, so that it can be accessed afterwards. This component communicates with the YAWL engine through the Interfaces A and B.

2. Organization Manager - Similar to the Workflow Manager, the Organization Manager is used to manage the users in the system. This component must also synchronize the users in our system with the ones created in the Resource Service, described in section 3.4. The synchronization is done using the user name, meaning that each user must have the same name both in the system and in the Resource Service. The only operation that cannot be synchronized is the creation and edit of a user since that part of the interface for the Resource Service is read-only. The implication of this is that the user must create his register with the Resource Service using its own web interface. This component communicates with the Resource Services using the Resource Client.

3. Data Type Manager - This component is responsible for managing the data types in our system. The main task of this component is to transform the data types representations that exist in our system to those that exist in YAWL. This task is rather straightforward since the data types structures are similar to the ones YAWL has.

4.3.6 Server Interface

This component is the interface through which other applications can communicate with the Workflow Service. Although the ideal interface would exchange HTTP messages with other applications, the present implementation works exclusively with the clients that uses the GWT RPC framework, explained in detail in section 4.4.3. The reason for this is that the framework unmarshalls and marshalls the messages automatically and transforms them into Java objects, which means that we do not have to do this ourselves. This is not a problem at the present time, since there are no applications that use our server beside our own.
4.3.7 Web Container

The Workflow Server is implemented as a Java Servlet and was deployed and tested on Tomcat 6.0. The reason that we choose Tomcat instead of other web containers such as Grassfish was mainly due to fact that the default installation of the YAWL engine automatically configures and deploys it on Tomcat. Since we wanted to avoid configuring it manually, mainly because there was no documentation regarding how this was done at the time, we decided to leave it on Tomcat. Although there is no obstacle impeding us from deploying the server on another Web Container, we decided to deploy it on the same web container the YAWL engine was deployed on to save us the trouble of managing multiple web containers.

4.4 User Interface

In the previous section we stated that we would add a workflow server. This leads to the problem of rendering the YAWL Workflow Editor, used to write YAWL specifications graphically, unusable in our architecture as it is. This is because the user interface communicates with the Workflow Services rather than with the YAWL engine directly. One way to overcome this would be to adapt the editor to work with the new server, and extend it to enable it to manage users and workflows. However, we decided to implement our own application that is a browser-based application, rather than a desktop application. This decision stems mainly from the fact that it’s more convenient to the users since they will not have to install the interface application on their computer. Another reason is that web applications are cross-platform, since there are browsers for all major operating systems, including portable systems.

Figure 4.5 is an illustration of the high-level architecture of the user interface. The architecture of the user interface consists of a central module called “Interface Manager”. It manages all the data within the interface, and also broadcasts the changes made to the different interface modules, such as the “Workflow Editor”, used to create and edit workflows. These modules are built by using a series of common elements, such as windows, forms, text fields, and toolbars, which are included in the “Interface Elements” module. Most of these elements are composed using the widgets present, in one of the GWT’s libraries, and some are animated using the “Animation Engine” to do the animation, referred in sections 4.4.1 and 4.4.2, respectively. Finally, the communication with the Workflow Server is done by using the “Server Interface Client”. The rest of this section will focus on describing these components.
4.4. USER INTERFACE

4.4.1 GWT toolkit

The interface itself was programmed using the Google Web Toolkit (GWT) [29] and the EXT JS [21]. The main advantage of using these toolkits is that they provide high-level interface elements such as windows, forms, trees and grids. They also provide the abstractions needed to easily implement some functionalities such as sending AJAX messages and implementing draggable objects. There are many other toolkits that we could have used such as Dojo [19] or Prototype [58]. However, we choose GWT because it is the only one that allows us to program in Java rather than in Javascript, thus providing more robust abstractions. This is because GWT compiles the Java code into the Javascript code before deploying the application. GWT also has a library called GWT-Ext, which is essentially a wrapper library that compiles the Java classes of the library into Ext JS code.

4.4.2 Animation Engine

In order to improve the overall quality of the interface an animation engine is used. An animation engine is an library that contains many functionalities that are generally used in creating animations, such as classes that create animations based on a series of images. However, not many engines exist for javascript, and none for GWT. This is why we implemented one with limited functionality ourselves.

The basis for implementing animations is to have an infinite loop that updates a static element
CHAPTER 4. SYSTEM ARCHITECTURE

and redraws it at short intervals, giving the illusion of movement. This loop is called animation loop [22] and is commonly used in video games. This loop is generally placed inside its own thread so that it does not block the rest of the program. However, this cannot be done in our application since Javascript is single-threaded. We solved this by using timed events that Javascript possesses to emulate an animation loop. This is done by creating a function/method that before finishing creates an event that calls itself, and is triggered a few milliseconds afterwards. The frame rate in our system is set to as low as 20 frames per second because a higher frame rate might cause some performance problems in case the interface is used on a machine with lower specs.

4.4.3 Server Interface Client

The Server Interface Client module handles the communication between the client and the server. This is done using AJAX messages. Using AJAX is convenient since it does not require the browser to refresh every time new data from the server is needed. This is essential to our interface, since having to reload the whole workflow every time we need to fetch some data from the server, would render our application unusable for larger workflows. Moreover, we use GWT’s RPC mechanism that allows the interface to invoke server-side code. To do this we had to implement our server as a GWT RPC Service, which is essentially a wrapper class that contained all the methods that could be invoked by the client. The main advantage of using this approach is that complex classes such as workflows that contain a list of nets, which in turn contains variables and nodes, can be sent to the server directly, leaving the serialization and unserialization of those classes to be performed transparently by GWT.

4.4.4 Interface Elements

Many interface elements that are used in the user interface can be reused in other components. Some examples include the Query Editor, which allows the user to write queries and the Item Selector, that allows the user to select an item (data type, workflow, node ...) from a filterable list of item obtained from the server using AJAX. The Interface Elements module is a library with interface these interface elements that are used by many components of the user interface.

Most of these elements are implemented either by extending widgets from the GWT toolkit or by aggregating these. For instance, the Item Selector is a window with a form that has two fields: a trigger field that allows users to type in the search parameters and a grid where the results are inserted.
4.4. USER INTERFACE

4.4.5 Desktop

The only module whose elements are not placed in windows is the Desktop module. Presently, the Desktop module only has the Desktop Toolbar and the Background component, which is only used to place a background image. As can be seen in figure 4.6, the Desktop Toolbar is a set of items at the bottom of the screen that allows other modules such as the Workflow Editor to be launched. Although the Background component is presently only used for a background image to be placed, the premise around it’s creation is for windows of other components to be minimizable to icons and placed in the background. The background object could then be serialized and stored in the server and reloaded the next time the user logs in.

4.4.6 Task Queue

This Task Queue module is used by the user to manage his tasks. The logic behind this is explained in section 5.6.3. This module presents a list of available tasks that are attributed to the user, and allows him to complete then by presenting a form that he has to fill to complete each task. The list is obtained asynchronously from the server and the form is constructed based on the specification passed to the node that generated that task.

4.4.7 Item Search

The Item Search module is used to search for various items in the system. This is used as a means for searching existing workflows and opening the Workflow Viewer. This module is basically a particular instantiation of the Item Selector described in 4.4.4.
4.4.8 Workflow Viewer

The Workflow Viewer module is used to manage existing workflows. This module has a component that allows the different versions of the workflow to be view and can load them using a Workflow Editor. Furthermore, this module also displays the instantiations of the workflows that are running or that have ended. It also displays the events that occurred during the workflow allowing them to be monitored, which is described in section 5.3. Although the list of events can be used to monitor a workflow’s case, this would be ideally presented in a form of a graph similar to the one in the Workflow Editor and allows the user to see the event like a node starting in the graph itself. This could be done by parsing the messages of the events put by the system, by the order they occurred, and reflect these event in the graph.

4.4.9 Workflow Editor

Workflows are specified graphically using the workflow editor, which is illustrated in figure 4.7. The Workflow Editor is composed by many components. The components that are initially visible are the Toolbar, the Overview, the Console and the Net Panel. The following list provides a description of each of these components:

1. Toolbar - The toolbar provides a variety of options, including the creation of items (like nets, nodes, and variables), the deployment and execution of workflows.

2. Overview - This component provides a overview of the structure of the workflow. These are presented using a tree view perspective, where the root is the workflow. This root’s children are the nets of the workflow and each net node has 3 nodes for nodes (workflow nodes), edges, and variables, respectively.

3. Console - The console is used to provide feedback about the actions that were performed.

4. Net Panel - The net panel provides a graphical view of the workflow’s structure. Users can use the panel to manipulate the nodes within the workflow. YAWL is a graph-based language so the best choice was to use a graph library, such as Jgraph [51] which was used in the YAWL editor. One of the problems of using the GWT is, however, that Javascript libraries cannot be used unless they are wrapped inside a wrapper library, much like GWT-Ext, so our choices were limited. The one library was jsViz [38], which had a wrapper library for GWT. However, the project was in stasis, so we decided to avoid it. The library we used was called gwt-diagram [7], which we used to draw the edges/connections between nodes within the workflow. The nodes themselves were implemented by combining the drag and drop functionalities of Ext and the image manipulation functionalities of GWT.
4.4. USER INTERFACE

Figure 4.7: Workflow Editor Interface

During the creation of workflows, other components might appear. These are generally new windows that pop up, to perform certain tasks. The following list describes some of the most relevant components that need to be used in a workflow:

1 Item Creator - The item creator provides a list of items that can be created in the workflow, and their description. The items are organized into categories, which can be divided into other categories. The first category is whether an item is a net, node, or variable. The node category can be further divided according to the type of interaction that node has, such as Condor, User Interaction, or HTTP.

2 Node Editor - After creating a node, it can be edited using the node editor. This is essentially a form that allows the user to change the attributes of the node. Because there is so much information in a node, this form is divided into tabs. The tabs that are present depend on the type of the node, but the most general tabs include: (1) the general information tab, which presents the general information about the node; (2) the input and output mappings which present a editable list of the input and output mappings of the node; and (3) the predicates tab that allows the user to define the control-flow aspect of the node, that is, what edge or edges are followed after the node has completed it’s execution.

3 Variable Editor - This component is responsible for editing the variables of the workflow.
Like the Node Editor, it is a form where the information of a variable can be inspected and edited.

4.4.10 Interface Manager

The coordination between the different components is done by the Interface Manager. This module manages the internal representation of the entities in the interface, such as workflows and users, and the creation, edition and deletion of these entities are all done using this module. Whenever new entities are created, or edited in the internal representation, this module is responsible for communicating the change to the rest of the components.

Figure 4.8 displays an example of an interaction between the manager and its component, when a new node is created, the Interface Manager must communicate the creation of this node to the various components that require this information. In this case, these would be the Workflow Editor’s net panel, console and the overview. The net panel and overview would use this information to create a graphical representation of the node in the graph and tree, respectively and the console would display the message of the node was created successfully. Similarly, when someone edits a node in the net panel, for example changing the label of the node, the node is changed in the internal representation of the manager and sent to the other components so that they can react to the event. It is important to notice, that although the Net Panel which is the announcer of the event can update the node right away, it should also wait for the event announcing it before updating because the request to update a node might be turned down by the Interface Manager, in which case it would send a event announcing that a failure in editing the node instead. This is also applied to the rest of the events.

In order to allow new components to be created without changing the Interface Manager, the communication between this module and the rest of the components follows a publisher-subscriber structure. The Interface manager possesses a set of listeners to whom it reports all events. New components that wish to receive these events must register with the Interface Manager and define what to do with the events they want to listen to. For instance, to implement the Workflow Overview component we have to register the component with the manager and implement the handling methods for the events regarding the creation, selection, edition and deletion of nets, nodes, variables, and edges.

4.5 Summary

This chapter provides the architecture of our workflow management system. We also present alternatives to this architecture, and explain why those were not chosen.
Figure 4.8: Interface Manager Communication protocol
To summarize, our architecture comprises of an user interface, a workflow server, and the Custom Services. The user interface allows users to design and manage their workflows graphically through a web browser. These workflows are stored in a server and converted to YAWL specifications before being submitted to the YAWL engine. The YAWL engine is responsible for the actual execution of the workflows. The services that can be called inside the workflows by the YAWL engine are implemented as Custom services and are registered into the YAWL engine. Our workflow model defines workflows as a set of nets, which are in turn, aggregations of variables and nodes. These nodes have input and output parameters and mappings and are connected through edges.
5

System Implementation

The previous chapter gave a high-level view of the modules within our systems architecture. In this chapter, we will describe how some features of our system are implemented.

5.1 YAWL conversion

In order for the workflows of the system to be executed, we have to create a specification in YAWL that is equivalent to the workflow of our system. As mentioned in section 4.3.2, the Marshaller is the component that is responsible for converting our workflows into YAWL specifications.

The conversion is done in three steps. First, some nodes are inserted into the workflow for monitoring purposes, which will be described in section 5.3. Then the composite nodes are transformed into a set of atomic nodes. And finally, the resulting workflow is converted into the YAWL specification.

5.1.1 Node Expansion

We explained in section 4.2 that nodes are divided into atomic nodes and composite nodes. While atomic nodes have a direct mapping to YAWL specification elements, composite nodes are simply combinations of atomics nodes, which are, in a certain way, workflows that use the atomic nodes. Because of this, it is necessary to transform these nodes into a combination of atomic nodes. This can be a simple task of replacing the composite node by atomic node. As an example, if we wanted to expand a Web Service Node, we could simply replace this node with a YAWL Service Node that references the Custom Service ”yawlWSInvoker”, and pass the same parameters that were passed to the Web Service Node. However, we generally want to do something more refined, such as adding other failover nodes that allow the handling of the errors that might occur. This can be done by adding a user interaction node and executing it if an error occurs. Furthermore, we want to encapsulate these nodes into another net so that we only have one expansion for the Web Service nodes. To do this, instead of converting the Web Service nodes to the nodes described above we convert every Web Service
node into a Subnet node that references that new net. The new net then is placed only once in the workflow. An illustration of the result can be seen in figure 5.1.

5.1.2 Conversion to YAWL Specification

After converting the workflow until only atomic nodes are left, the conversion to YAWL is done straightforwardly, since most entities in the workflow model are directly converted to YAWL. “Workflow” entities are converted into “specification” elements where their “uri” attribute is obtained using their name, namespace, and version. The “metadata” element is also obtained using the workflow’s attributes. The “Net” entities are converted into YAWL decompositions of the type “NetFactsType”. The variables of the net are converted into “inputParam”, “outputParam”, and “localVariables”. The nodes are the only elements that are not done in a straightforward manner, since the information of the node is defined in two parts in the specification. First, there is the decomposition of the type “WebServiceGatewayFactsType” that describes the arguments it takes and the YAWL service to that it refers to. Secondly, the “task” element describes a particular instantiation of that decomposition, namely its control-flow and data-flow. These two are connected by the “decomposesTo” element of the “task” element. This is problematic from the object-oriented programming point of view,
since during the generation of the specification, the “task” element is generated first and the 
object for the decomposition element has not been created yet, and cannot be referenced. 
However, this is not a serious problem since the decomposition can be created right away but 
stored in a list and placed later.

5.2 Workflow Importing

One of the main goals of our system is to allow the reuse of NLP modules. This applies to 
workflows as well, since they can be viewed as modular tasks and used by other workflows. 
There are two problems with using other workflows in a workflow. First, the referenced 
workflow might not be deployed. Second the referenced workflow might be modified, making 
the workflows that reference it fail. The following sections mention the node that is used by 
the user to reference another workflow, and how the 2 problems are solved.

5.2.1 Workflow Node

The Workflow Node invokes another workflow when it is executed. The input and the output 
arguments of this node are those of the workflow that is references, which are in turn the 
input and output variables of it’s root net. The workflow is specified by the name, namespace, 
and version, but the interface makes specifying the workflow easier by presenting a filterable 
list of workflows and letting the user choose the one he wants.

5.2.2 Workflow Specification Injection

To solve the first of the problems mentioned above, we decided to embed the referenced 
workflows into the workflow itself. This makes the referenced workflows be deployed with the 
workflow. To do this during the expansion of Workflow Nodes, the referenced workflow is 
added to the workflow by inserting its nets in the workflow and replacing the Workflow Node 
by a Subnet Node referencing the root net of the referenced workflow.

5.2.3 Workflow Versioning

The second problem is solved by creating a version system for the workflows. Every time a 
workflow is saved, a new version of that workflow is created. When a workflow references 
another, it will actually reference a version of that workflow, which means that the referenced 
workflow will not be modified. This way, users will not have to guarantee that all the workflows 
that reference his workflow still work after making alterations. In case the user wants to use
the updated version of the workflow, he can change the version of the workflow in the Workflow Node. Furthermore, having a versioning system has the additional advantages of having a versioning control system, which include the ability to go back to a previous version of the workflow. Also, in a situation when two or more users edit the same workflow at the same time, their updates will not be lost.

5.3 Workflow Monitoring

Instead of simply running a workflow and presenting its results, we wanted to allow the monitoring of the workflow during the execution. From the start to the end of a workflow case, our system allows the user to know which nodes were executed first and what was the state of the workflow prior and after the execution of each node. This section describes how this feature is implemented.

5.3.1 Event Logging

Monitoring of the workflow’s progress is done in a similar way to the Windows Workflow Foundation’s Tracking Service [48], in which the monitoring of the workflow’s instances is done by storing a series of events that are emitted during their execution. To be able to do this, we would have to get the YAWL engine to inform the server at certain points of the workflow instance. There were two ways to do this. The first way is fairly straightforward: we simply program our Custom Services to notify the server before and after actually performing the work of the service. This would allow us to know when a task starts and ends, but we would not be able to track other events such as the start and end of the workflow. Furthermore, we would have to add additional code to every YAWL service, which would affect our system’s extensibility. A better way involves implementing another YAWL service that is used to log events and place them in the key points we want. This has the advantage of allowing the user to use the service to define his own key points and at any point in the workflow.

There was another way to monitor workflows, where we would inquire the YAWL engine about the existing Work Items and using their states to tell whether they are completed or not. However, this method would also not allow the user to define his own key points and, furthermore, we would not be able to extend the data that is saved along the Work Items without altering the YAWL engine.
5.3.2 EventLogger Service

Logging of the events is done using the EventLogger Service, which builds and sends an HTTP message with the information of the event to our server so that it stores the event in the database. To this end the service needs the URI of workflow server and where it is to send the HTTP message. It also requires the event type, which identifies whether the event was created by the user or automatically by the system, and the actual message that describes the event. Finally, in order to allow the workflow server to know which workflow’s case the event is related to, an identifier is sent. As identifier we used the identifier that the YAWL engine gives to each case it launches (“caseId”), which is unique for all workflow cases. On the server’s side, when a event is received it is tagged with a timestamp and also with a snapshot of the state of the workflow which consists of the values of every variable in the workflow so that the user can follow the progress of the workflow.

We could not have used the HTTP Service described in 5.6.2 instead of creating a new Service, because we the identifier of the workflow’s case can not be passed as a parameter to the HTTP Service since is only accessible at the level of the Custom Service.

5.3.3 EventLogger Node

To log an event in a particular part of the workflow we use the EventLogger node, which is essentially a composite node type that is decomposed into a YAWL Service Node that references the EventLogger Service. The input parameters of this node consist simply of the message that is to be associated with the event. The other parameters, the URI of the server and the event type are automatically filled in by the Workflow Server.

Aside from the events that are placed by the user, the system also injects the workflows that are submitted with EventLogger nodes at key points, thus allowing it to monitor the progress of the workflow.

First of all, two nodes are placed at the start and end of the workflow. Both nodes are used to track when the workflow effectively starts, but the node at the end of the workflow also allows us to get the output values of the workflow. The main reason this needs to be done is because the YAWL engine does not keep the workflow cases after they are finished running. This means that it is not possible to request the data of the workflow case afterwards, so, by placing a node before the output node, we can get these values since the Workflow Server takes a snapshot of them when a event is stored.

Similarly, two nodes are also placed at the start and end of each node, which allows the monitoring of when a node starts and ends. To do this, we actually have to create another
net with the nodes to do the logging and the node to that actually performs the task (see figure 5.2). This is because, if we were to simply add the logging nodes before and after that node, the edges that lead from the node to the following nodes, in the case of the example nodes “Node3” and “Node4”, would be transferred to the node that logs the end of the node, which is shown in figure 5.3. The problem with this transfer of edges is that the predicate query that does the routing of that node cannot use the output of the node, since the transferred edges do not stem from it, and will access the output of logging node instead. This is why we add a net that has the same input and output arguments of the node to log, and replace the node to log with it. That way the net will take the same arguments and produce the same results as the node to log. Also the structure of the workflow remains the same, since the node to log is simply replaced by an equivalent node.
Figure 5.3: Logging workflow nodes without a wrapper net
5.4 Condor Support

Grid computing allows sharing of CPUs to provide massive processing power. This kind of computing power is driving a new evolution in industries [8]. The field of NLP is not an exception, which is why it is essential for our system to allow the access to grid-enabling technologies such as Condor, during the execution of a workflow. This section describes the details of the Condor related features of our system. Condor can be accessed using its Web Service interface called Birdbath (see section 5.4.1). To access this interface, we defined a Custom Service called Condor Service(see section 5.4.2 that can be used using a set of Condor Nodes(see section 5.4.3)).

5.4.1 BirdBath

BirdBath [14] is a Web Service interface for Condor. It provides a set of Web Services, each defining an interface to access a different daemon of Condor. The different daemons have distinct responsibilities, the “Condor Master”, for instance, is responsible for collecting all the information about the status of a Condor pool. In our case, we will be using the “Condor Scheduler” which is the responsible for submitting and managing jobs in Condor. There are other interfaces for accessing Condor, such as DRMAA [60], but we chose BirdBath because it is included on all releases of Condor after version 6.8 [59] and can be activated simply by enabling it in the condor configuration. This saves the users the trouble of installing extensions for Condor.

5.4.2 Condor Service

The interaction between the YAWL engine and the Condor System is mediated by the Custom Service, Condor Service. This service uses the BirdBath API to send commands to Condor. Currently, there are 3 commands implemented in this Service:

- Create Job - Creates a single job in condor.
- Get Job Info - Queries the information available for the job.
- Get Job File Content - Which gets the content of a file that the job writes to.

The multiplexing of the command to use is defined by the parameter “CondorOperation”, which must be passed to the service along with the parameters for the command itself. Users will not have to worry about this parameter though, because this is done automatically by the Condor Nodes. The only exception occurs when the YAWLServiceNode is used to call this service.
5.4. CONDOR SUPPORT

5.4.3 Condor Nodes

For each of the commands available in the Condor Service, there is a composite node that hides the complexity of calling the service natively. The nodes “CondorCreateJob”, “CondorGetJobInfo”, and get “CondorGetFile” are used to create jobs, get information about a job, and get a job’s file, respectively.

5.4.3.1 Submitting Jobs

The CondorCreateJob Node, as the name implies, creates a job and submits it to Condor. This node works in a similar fashion to the “condor submit” command, which receives a submit-description file, where the attributes of the submission are described. These attributes include the command to execute, it’s arguments, the in, out and err files, the requirements under which the job can be executed, and the priority of the job. In the case of the CondorCreateJob these attributes are used as input parameters of the node. The node returns the identifier of the job and the identifier of the cluster where the job is submitted to. These are used as a reference to the job for other nodes.

5.4.3.2 Querying Jobs

After a job is submitted, the CondorGetJobInfo Node can be used to check the status and progress of the job. This node receives the identifier of the job (cluster and job id), which are provided upon the creation of the job. The output of this node is similar to what one would get with the command “condor q -l”, which lists more than 30 attributes for each job. Among these, one useful attribute is the state of the job, which can be used to check if the job has finished or if an error has occurred and the job has been put on hold. Each of these attributes is passed as an output parameter by the node.

5.4.3.3 Fetching The Output

The output of Condor Jobs are generally written in files, and accessed by other jobs. This can be done by using the CondorGetFile node, which gets the content of a file in a job’s Spool. The Spool is a temporary directory that Condor creates for each job that is submitted and used for temporarily storing the files that are produced in the job. This directory is erased after a job is removed, which means that some caution is required in order to make sure that they are not deleted before accessing them. To avoid the problem of not fetching a job’s files before they are removed from the queue by the Scheduler, the jobs that are submitted by default with the attribute “LeaveJobInQueue” with the value “FilesRetrieved =?= FALSE”,...
which means that the job is left in the queue until the files are retrieved. If the user does not wish this behavior, he can define others by overriding this attribute or any other when submitting the job in the CondorCreateJob node.

5.4.4 Job Monitorization

There is no callback mechanism in Birdbath that will inform interested programs when a job is finished. Knowing when a job finishes is important since all the tasks that are dependent on that job need to know when they can start. To do this, we use the CondorGetJobInfo node, to get the state of the job and check whether it is completed. While the job is still not completed, we wait a determined amount of time (it is recommended to wait over 20 seconds in order to not overload the Condor Environment) before checking again. The waiting can be done using the Wait Node defined in section 5.6.3. This will be repeated until the job is completed, after which the workflow proceeds to the next node. Figure 5.4 illustrates the part of the workflow that is described above.

5.4.5 Error Tolerance

Sometimes, jobs in Condor can be put on hold if, for instance, the job is not correctly submitted because some file is missing or some parameters are incorrect, or if the Condor System cannot be reached. If this happens, the user should not have to restart the workflow. This is why a error-handling node is needed. There are two types of errors that are considered: a job is put on hold for some reason, and the Condor System cannot be reached.

We can check when a job is put on hold by using the CondorGetJobInfo node in order to check the state of the job. If the job is on hold, the error-handling node is activated. The error-handling node is simply a User Interaction node, requesting the user to provide some data regarding whether, the job should be rerun or ignored. Furthermore, the user can also change the attributes of the job, such as the command and the arguments.
5.5 USER INTERACTION

Checking whether the Condor System is down requires the Condor Service to provide some data regarding whether the remote invocation to Condor was successful or not. This can be done by adding an output parameter to the response of the service. If we detect a failure using this parameter, in a similar way to the case above, we activate an error-handling node, which will allow the user to specify if he wants to retry or ignore the node. The main difference in comparison with the situation above is that every Condor Node can fail. Because of this, need to include the error-handling node in every one of them.

5.5 User Interaction

There are many times when, during a workflow, the system might need intervention from a human. The workflow may suffer from an error and need to request the user for instructions on what to do next or users can manually do the work of some components that are yet to be implemented. Our system needs to enable nodes that, when executed, will require a user to perform a certain task, such as filling a form. This chapter describes the user interaction capabilities of our system.

5.5.1 YAWL Resource Service

As described in section 3.4, the YAWL resource service provides a model for users and organizations as well as the means to allocate them jobs. Thus, it would benefit us to use this service, since it would save us the trouble of implementing a component to manage the allocation of jobs to users. Like the YAWL engine, the Resource Service provides a HTTP interface allowing exterior applications to communicate with it. However, unlike the YAWL engine, this interface only provides read-only operations for managing users. This means that users cannot be created or edited but only viewed. Another problem is that it also does not provide a way of passing the parameters to a job, so for instance, if we had a node that would require the parameters x and y from the user, we would not be able to provide it using the HTTP interface. We will discuss how these problems are handled in the next two sections.

5.5.2 Creating users

Since it is not possible to create users in the resourcing service by an external application, we decided users are registered using the Resource Service. This is convenient, since it saves us the work of implementing some functionalities, such as a login system with password encryption and a interface to create and manage users, since these will be delegated to the Resource Service. We will still have a model for users in our own database, which is synchronized with
their, for attributes not present in their service that we might need. Although, presently our user model is not used, except for defining the creator of a workflow, we might need it to define, for instance, permissions regarding access to each workflow.

5.5.3 Completing Jobs

YAWL’s Resource Services also does not allow passing the parameters of a job. To overcome this, we will be passing the arguments and completing the task directly into the YAWL engine. This can only be done if the job has not reached the “started” condition. This is because after a job starts, the Work Item that is associated with it is checked out of the YAWL engine, which basically means that it is erased from the engine. So, instead of letting the Resource Service checkout the Work Item, we will be checking out the Work Item ourselves, pass the parameters and check it back in. There is a small drawback with this, because the Resource Service is oblivious that we checked out the Work Item, so the job is still left in the Resource Service. However, when the whole instance is finished all the jobs associated with it are erased, so it is not a big problem.

5.5.4 Allocating Jobs

The previous section discusses how jobs that are allocated to a user are completed, but does not talk about how the jobs get in the queue in the first place. The jobs are allocated using the Nodes whose types are expanded to the Resourcing node, which reference the Resourcing Service. At the moment, the only node of this type is the Prompt Node, which schedules a job for a specified user to provide some parameters, which are the output parameters of the node. The node has two input parameters, which are the message to be displayed to the user and the specification of the fields of the form to be presented to the user. The second parameter is an XML string, where the root’s children are “parameter” elements, with an attribute “id” that references the parameter that is associated with it. Inside the “parameter” element, the user needs to specify the type of field that it wants for that parameter. Currently, there are only two types of fields: text fields and combo boxes. The former is chosen by putting an empty “text” element in it, while the latter is placed by using a “select” element, with “option” elements as children. “Option” elements has a attribute “value”, which is the value the respective parameter gets if chosen, and the text of the option is represented by inner text of the element. Figure 5.5 gives an example of how a form with a text field for name, and a combo box with “Portugal”, “Japan”, and “China” as options.
5.6 Other Services

5.6.1 Web Services Support

The first nodes that were implemented were Web Service Nodes. This is mainly because it is a service that comes with the YAWL package. The Web Service Node lets the users define the input and output parameters, which are passed to the Web Service invocation. There are three required parameters, which are the location of the web service, the port and the operation to invoke.

One possible extension for this module is to have the interface parse the WSDL file and find out the operations and ports, allowing the user to select the desired one. This way, the interface can place the input and output parameters automatically, making this process easier and less error-prone.

5.6.2 HTTP Support

Most web pages can be accessed using the HTTP Request node. This node is used to send “get” or “post” messages to a certain URL, which are both specified as arguments. Aside from these, more parameters can be defined and are used as the parameters of the HTTP request. The Custom Service that implements this function uses the “java.url” package to send the requests. This node is useful when the server is implemented to support an HTTP protocol, or if we want to access the contents of a web page.

5.6.3 Scheduling Jobs

Some times we want a certain node to be launched at a certain time or simply wait a few seconds before proceeding. This can be achieved using the Scheduler Service, which checks the Work Item out and waits until the desired time passes and checks it back in. This is
done using “java.util.Timer”, which allows the process to be notified when a certain time has passed. The only node that uses this service in the current version is the Wait Node which allows the user to specify how much the node must wait until it completes.

5.7 Extending Services

It is likely that more services will eventually be needed in the server to satisfy new needs. In this section we discuss the different ways to achieve this.

The easiest way to extend services is to simply use the existing services to call new ones, for instance, deploying a Web Service to send SMS and using the Web Service Node to invoke it. The main advantage of this approach is that we can implement the service on a platform that we are familiar with instead of implementing a YAWL Service. The problem with this approach is that the service will not receive most notifications that YAWL Services do. For example, YAWL Services are notified when the node is cancelled by the user and knowing this can allow the service to undo its work if necessary.

The way we recommend is to use the YAWL Service Node. As mentioned in section 4.2, the YAWL Service Node allows the user to associate the node with a particular YAWL Service. Following this approach, the user would have to implement the YAWL Service, which is well documented, and reference the service using the YAWL Service Node. This approach allows the service to receive all the events given by the YAWL engine, and does not require much effort.

Note that in both methods above the user can wrap the service inside a workflow so that it can be reused with less effort by other users.

Finally, the user can manually write the code to create another node type, the same way the existing node types were created. This method requires more effort since the user needs to implement both the interface and the node type. However, the benefit of doing this is that it allows the user to mold the interface of the node the way he wishes and gives him control of the process of the decomposition of that node. For instance, the new node could be decomposed into a Web Service Node or a Condor Node depending on the input parameters passed to the node.

5.8 Summary

This chapter covers the various implementation details of our system. We discuss how the conversion to YAWL specifications is done, which is a three step process, starting with the
addition of nodes to monitor the workflow, the expansion of nodes and ending with the actual
generation of the YAWL specification. We also present the various nodes/services of our
system. These include (1) the Workflow Node that is used to embed other workflows in our
own, (2) the EventLogger node, which is used to monitor workflows, (3) the Condor nodes,
which can be used to submit and manage jobs from Condor, (4) the User Interaction Node
that allow the user to participate in the execution of the workflow, (5) the Web Service node,
which invokes Web Services, (6) the HTTP Request node that sends HTTP requests and
finally (7) the Scheduler Node that allows the timing of the execution of nodes. This chapter
ends with suggestions regarding how new Services can be added to the system.
This chapter discusses the strengths and weaknesses of our system.

The most obvious advantage of our system is that it allows us to build our applications graphically. This has the benefit of providing a more accessible framework for programmers and non-programmers to define complex applications. Moreover, it provides a visual definition of the system that can be presented to the stakeholders of the application. The main problem with the interface that limits part of the usability of the framework is the fact that there are two elements of the interface that require the user to be familiar with the XQuery language that the YAWL engine uses. The first element is the mapping between the variables and the parameters of the components and the second element are the predicates or conditions that control the control flow aspect of the application. However, the interface helps to a certain extent, in that it provides the mechanisms to generate the code for accessing variables and parameters of the application, allowing simple queries such as the direct mapping between the output of a component to a variable achievable without knowing the programming language.

Another advantage of our system is that the YAWL engine that we use to execute the workflows allows various workflows that other workflow engines do not. Some examples include workflows containing cycles and the ability of one branch in the workflow to split into an arbitrary number of branches. There were also some features in YAWL that are not supported in the current version of our system, namely, the creation of multiple instances of a component and the definition of cancelation sets, which were not implemented in our system, since they are currently not needed to test the system.

A more philosophical impact of the system is the fact that it forces the users to separate the roles of software engineer and programmer. The difference between these roles is that the software engineer will use our system to design the whole application/workflow in a highly abstract fashion, and leave the implementation of the individual components of the workflow to different programmers that will use our system to submit those components. The submission of the components can be done in various ways, which are described in section 5.7. The advantage of submitting those components into our system is that they can be used in other applications as well, which is useful in the field of NLP since many components, such as part-of-speech taggers, are used by various NLP applications. This allows saving some work in new applications since, as more applications are build into our system, more components
that are generally used in this or another field will be registered, and can be used simply by searching the existing components that are registered in the system and inserting the corresponding node in the workflow.

The workflow monitoring feature of our system allows not only the debugging of applications but also their evaluation. For instance, if we want to improve the efficiency of an application we can use the workflow monitor’s information to determine the components that are the bottlenecks of the whole application.

Finally, our system also allows programmers to create error handling mechanisms. An useful example is to allow a Condor Job that goes wrong to be recoverable, instead of having to repeat the whole application that might have been running for hours. By combining different services, we can create interesting error handling mechanisms, such as sending a e-mail and notifying certain users, which can be determined at run-time, that something unexpected has happened and go to a service to receive and parse a reply from that user on what to do next. Although, this is not entirely exclusive to our system, and can be done in most programming language using libraries to make it easier, we still see many applications that do not handle errors since when programming those applications the programmer does not want to waste time to make a error handling module. In our system, we would only have to implement the component that sends a e-mail and parses the reply and sends the required parameters to the output, and all users can then simply insert that component to their application.
Conclusion

In this work, we presented a system that allows the implementation and execution of complex NLP applications in a high-level of abstraction without the need to have a deep understanding of programming, using workflows.

The execution of the workflows is delegated to the YAWL engine, which uses the YAWL language that has a high level of expressiveness allowing the definition of complex workflow constructs such as arbitrary cycles and also mechanisms for cancelation and advanced synchronization, thus surpassing the expressiveness of other workflow engines such as BPEL. Furthermore, the YAWL language is based on Petri-Nets, which is simple to process, since a workflow is simply defined as a set of nodes and edges.

The current version of our system allows the definition of workflows in a graphical editor. These workflows are submitted to a workflow server, where they are stored, managed and scheduled for execution. By storing the workflows in a server, all the work that is submitted to the server can be reused in other applications with little effort. Additionally, the system also keeps track of the various versions of the workflows to allow users to freely update their workflows without worrying about making other projects that reference it unusable.

The communication between the engine and external applications is done through a group of loosely coupled services called YAWL Services. Through these services, the engine is capable of communicating with multiple environments, including Web Services, HTTP, and Condor. The system also allows the implementation of additional services, which can be registered with the system without having to recompile or restart it.

Finally, the system also provides workflow monitoring mechanisms that allow the state of the workflows to be known. This is done through an event-based mechanism, where whenever a node starts and ends, a event registered. Furthermore, the user can define failover behaviors for his components allowing them to be recovered without restarting the workflow.
7.1 Future Work

7.1.1 Permission and user management

Currently, there is no permission system for the workflows. This means that all workflows that are submitted to the system are accessible to every user to view, run, and edit. A user may want to have exclusive ownership of an application or want to allow only the users who pay him to be able to execute it. In a structured organizational environment, it is required that the system models the roles within the organization and their respective permissions. One way to do this is to extend the YAWL’s Resource service that our system uses to attribute work to users registered within the system. This service already defines the organizations model allowing the definition of permissions for each role and user, and we would only have to link these entities to our workflows.

7.1.2 Workflow Validation

A way to improve the usability of the workflow editor is to allow the user to check whether the workflow is valid. Since the structure the language of our system is equal to YAWL, we can verify the correctness of our systems by checking for the properties of soundness and completeness, which is proved in [33].

7.1.3 Implementing Additional YAWL Services

The implementation of additional YAWL Services and thus increasing the variety of choices for users when creating workflows: services such as a service for communicating with Globus, or simple services like sending and receiving emails.

7.1.4 Support for BPEL

Although we chose the YAWL engine to execute the workflows, because BPEL is the most popular choice for workflow enactment, we should also support the execution of our workflows in a BPEL engine. This would involve the conversion from YAWL to BPEL. The main problem with this is that YAWL is more expressive than BPEL, which means it will be difficult to convert some workflows that have constructs such as arbitrary cycles or cancelation sets, because BPEL does not have any way to represent this. Thus, the work to fully convert the YAWL specification to BPEL could be difficult and very little work as been done in this subject. Another feature related to BPEL would be the ability to run BPEL specifications by converting them to YAWL. This conversion is easier and is documented [5].
7.1.5 Improve Workflow Monitoring Interface

The current monitoring interface consists simply of a list of events and their respective timestamps. A more natural and intuitive way to present this information would be by presenting the workflow itself and show the changes in the workflow as the time advances. An example of the illustration of an event in a workflow illustration would be the events regarding start and end of workflows. These can be parsed to identify which node has started or ended and illustrate the change by doing something as changing the color of the node.

7.1.6 Concurrent Workflow Edition

Currently, when two people edit the same version of a workflow and save it, the system will create two different versions of the workflow so that none of them loses their work. A more interesting approach would involve allowing both versions to be merged. This could be done at the Net level, which means that the system would have to check which nets were altered by each user so that a conflict would only occur if both users were working with the same Net in a workflow, and let them chose which version to maintain. The same could be done at a lower level where we would also check the nodes, variables, and edges.

Furthermore, the workflow interface could communicate regularly (every 10 seconds) with the server to check whether there are new updates of the workflows that are currently open and update these transparently for the user and notify only when a conflict has occurred.
Bibliography


This chapter will provide an example of the definition of an executable workflow using our system.

### A.1 Logging in

After loading the system’s interface/web page, the interface will first check whether the server is configured properly, and ask the user to configure it in case it is not. Provided that it is, we will see an interface with a single button in the toolbar, which opens the login window upon clicking on it. We can see the interface in figure A.1 and figure A.2. Finally, the instructions to configure and create users is provided in the appendix B.

### A.2 Workflow Definition

In this section we will describe the creation of a workflow.

#### A.2.1 Creating a new workflow

After logging in, the first step to take is to open the Workflow Editor, which can be done by pressing the Workflow Editor button that appears at the toolbar after logging in. This will open a new window where workflows are defined, which is show in figure A.3.

To create a new workflow, we will right-click on the Projects node at the left and a menu will pop up, where we can choose to either import an existing project into the project list or create a new project (figure A.4). We will chose to create new project, by doing so a new window asking which kind of project to create (figure A.5). Presently, we can choose either to create a Workflow or an Audimus Specification. We want to create a workflow we will chose the former and submit. The interface will now ask for the name and namespace of the workflow (figure A.6). Finally, a new workflow will be created. We can see in figure A.7, that in the console a message confirming the creation of the workflow and in project overview to the left we can see a new icon with the name of the workflow.

#### A.2.2 Adding a root net

After a workflow is created we should define a root net that will be executed when the workflow is launched. To do this, we need to right-click on the project icon in the left, which will open
Figure A.1: Guest interface

Figure A.2: Login Window

Figure A.3: Workflow Editor
Figure A.4: Projects Menu

Figure A.5: Projects Creator Form
Figure A.6: Create Workflow Form

Figure A.7: Workflow Editor - Create Workflow
A.2. WORKFLOW DEFINITION

a menu (see figure A.8). We will chose the to create a new net, after which the interface will ask for the name of the net (see figure A.10). After submitting, a visual representation of the net will be displayed. To set this net as root, right-click on the net icon to the left and click on the set root item. We will see that the name of the net will have “(ROOT)” appended indicating that it is the root net.

A.2.3 Adding new nodes

To add new nodes, we once again right-click on the workflow icon and chose to create a new item. This will show a new form where different items can be created (see figure A.11). Under the nodes folder, we can choose various types of nodes that can be created. To actually be able to execute this, we will create a PHP (PHP Hypertext Preprocessor) script that will simply return the message “helloworld” when a HTTP request is received. To call this script, we will use the HTTP Request Node. After choosing and submitting this node, it will ask for the properties of the node. These properties depend of the type of node chosen. For instance, the subnet node executes another net in the same workflow and, therefore, it asks for the net to be executed. In this case, we need to provide the label of the node and the parameters of the HTTP request. Since the script does not need input parameters, we will simply set the label to “hello” and submit the node. This will create a node in the workflow. Now we want to connect the node with the rest of the start and end nodes to make the workflow sound. This is done, in this case, by connecting the start node to the “hello” node and connecting the “hello” node to the end node. Connecting nodes is done by selecting the starting node by clicking on it, then clicking on the ending node with the shift button pressed. The result is shown in figure A.12.

A.2.4 Adding new variables

We will need to create a variable to store the response from the server. To do this we will once again right-click on the workflow icon and chose to create a new item. This time we will chose to create an variable from the list in figure A.11. This will present the form for the creation of variables (figure A.13). We need to provide the name, usage (whether it is an input, output, or local variable), and the type of the variable, which can be fetched from the server. We can optionally set a default value for the variable. After submitting, the new variable will appear in the variables folder to the left.

A.2.5 Setting parameter mappings

After creating a variable, we will need to set the mappings from the output of the HTTP Request node to the variable. This requires us to edit the node again, which can be done by double clicking on the node. The form to edit allows the definition of the input (figure A.14) and output (figure A.15) parameter mappings as well as the flow mappings (i.e. conditional routing). We will set the input parameters “method” and “URL” of the request to “get” and http://localhost/hello/hello.php, respectively, and the output parameter, response to the variable we just defined. This requires us to write queries in the XQuery language. Fortunately, the editor helps by providing the queries corresponding to the variables and
Figure A.8: Workflow Menu
A.2. WORKFLOW DEFINITION

Figure A.9: New Net Form

Figure A.10: Workflow Editor - Create Net
Figure A.11: Item Creator
output parameters of the node, which is shown in figure A.16. After all the parameters are
set, submitting the form will commit these changes.

A.2.6 Saving the workflow

Now that the workflow is defined, we can save it by right-clicking on the workflow icon and
select save on the menu (figure A.8). Since this is the first save a message will be displayed to
confirm that the workflow has been saved and that it is the first version.

A.2.7 Executing the workflow

To execute the workflow, we need to deploy it first, which can be done right-clicking on the
workflow icon and select deploy on the menu (figure A.8). If the specification is correct it will
present the message “success”. After the workflow is deployed we can execute it by right-
clicking on the workflow icon and select execute. A form will be presented to prompt the
input parameters of the workflow, which we do not need since we didn’t define any, so we will
just submit and the workflow will start executing.

A.3 Summary

This chapter provides an example of the usage of the system, namely, the logging in process,
various aspects for the creation of a workflow and the it’s execution.
Figure A.13: Create Variable
A.3. SUMMARY

Figure A.14: Node Editor - Input Parameters

Figure A.15: Node Editor - Output Parameters
Figure A.16: Query Editor
**B.1 Installation Instructions**

To install our system we will need the following items:

- A YAWL engine (Tested with Release Candidate 1 & 2), which can be downloaded at [http://www.yawl-system.com/](http://www.yawl-system.com/).

- A database (MYSQL 5), which can be downloaded at [http://dev.mysql.com/downloads/](http://dev.mysql.com/downloads/).

- A web container (Tomcat 5.5 or higher), which can be downloaded at [http://tomcat.apache.org/](http://tomcat.apache.org/).

- The NLPEditor.war file, with the actual system.

The instructions for the installation of the YAWL engine, database and web container can be found at their respective websites. After installing them you will need to deploy the NLPEditor.war file in the web container.

**B.2 Configuration After Installation**

After deploying the system into the web container you should be able to access the server at [http://example.com/NLPEditor](http://example.com/NLPEditor). You will see a loading message, indicating that the system is testing the server to check whether the current configuration is functioning correctly, and if not, it will give a notice of the items that are not configured properly (figure B.1) and present a configuration wizard (figure B.2). This could take from 2 to 4 minutes if the server is not configured.

Clicking “next” will guide you to the configuration of the database B.3, where you will be asked to provide the data corresponding to the database you wish to use. You can click the “Test Configuration” button to check whether the configuration is working correctly. The configuration parameters of the database are simply the parameters generally included in the Hibernate’s hibernate.cfg.xml file. You must make sure that the database you choose to use has already been created.

Here is an example of a configuration using a MYSQL database:


Figure B.1: Notice with unconfigured items

Figure B.2: Configuration Wizard
Figure B.3: Database Configuration
• hibernate.connection.driver_class - com.mysql.jdbc.Driver.
• hibernate.connection.url - jdbc:mysql://localhost/nlpwf.
• hibernate.connection.username - root.
• hibernate.connection.password - qwerty.

Clicking the “next” button will lead you to the YAWL configuration (figure B.4). Once again, you will need to provide the information corresponding to the YAWL engine that you wish to use. And again, you will be able to test whether the configuration is working. You must provide the URL where the YAWL server is and the username and password of a user registered in the YAWL engine. The YAWL engine’s default administrator has the username “admin” and the password “YAWL”. Note that you must have the YAWL engine up and running, otherwise the system will not detect the engine’s presence and consider it unconfigured.

Here is an example of a possible configuration:

• yawl_url - http://localhost:8080/yawl/.
• yawl_username - admin.
• yawl_password - YAWL.

After finishing YAWL’s configuration, click “next” and “finish” (figure B.5) to commit the new configuration.

B.3 Installing from source code

To install from the source code you will need to compile the source to produce the NLPEditor.war. Before starting you will need the following:

• You will need Apache Ant, which can be downloaded at http://ant.apache.org/.
• You will need to have JDK 5 or higher. This can be obtained at http://java.sun.com/javase/downloads/index_jdk5.jsp.

Afterwards, you can generate the NLPEditor.war file and the javadoc by prompting “ant” in the NLPEditor folder or “ant compile” if you do not want to generate the javadoc.

B.4 Importing projects into Netbeans

To compile from Netbeans, you will need to import source as a Netbeans project. To do so, select “New Project...” under the File menu, and choose “Web Application with existing
Figure B.4: YAWL Configuration
APPENDIX B. INSTALLATION GUIDE

Figure B.5: YAWL Configuration
sources” under the category “Java Web” and click “next”. Set the location into the location of the build.xml file located at the root of the source code and click “finish”. The project should now be imported into the Netbeans environment. You can build the project by right clicking on it and select the “Build” Item.

If you wish you can comment the “-init-taskdefs” and “library-inclusion-in-archive” targets from the build.xml file. These targets override the ones set by Netbeans, and are needed in order to be able to build the project independently from Netbeans.

B.5 Creating Users

To create an user you must first create a user in the YAWL’s resourcing service which can be accessed in YAWL. This can be done using the YAWL’s resourcing interface at http://example.com/resourceService. To do this you must log in (the default username is “admin” with the password “YAWL”) and click on the “Users tab”. The password and the username must match the ones you wish to have in the system.

After the user is created, you simply need to try to log in in the system using the password you wish to have and the server will create your profile based on the information in the YAWL’s resourcing service.

B.6 Installing the YAWL Custom Services

The source provides the following YAWL Services that must be deployed before using them. To do this you can compile them using the “ant” command at the root of their respective folders. This will generate the .war files that must be deployed at the same web container where the YAWL engine is.

After deploying them you must register these with the YAWL engine. This can be done using the YAWL’s resourcing interface at http://<webcontainerhost>/resourceService. To do this first log in as an administrator (the default username is “admin” with the password “YAWL”) and go to the “Services” Tab. The name and description that you give to the service is not important but the URI must point the uri of the YAWL service.