Universidade de Évora

Mestrado em Engenharia Informática

Workflow modeling using UML, Declarative Tools and WEB2.0

Rui Alexandre Rodrigues Gamito
<rgamito@lnec.pt>

Supervisor: *Salvador Pinto Abreu*
Co-Supervisor: *Luís Arriaga da Cunha*

Évora, June 12, 2008

This thesis does not include appreciation nor suggestions made by the jury.
Esta dissertação não inclui as críticas e sugestões feitas pelo júri.

Abstract

Manual translation of UML diagrams to programmatic code is tedious and error prone. Many CASE tools allow computer code to be generated from Class Diagrams, but fewer, if any, allow the transformation of Activity Diagrams (ADs) in executable and workflow defining computer code.
Our project aims at:

- Translating UML ADs into the ISCO programming language;
- Building a workflow execution system that runs the translations;
- Building a graphical tool to create and edit workflows, and visualize the executions.

Defending that UML ADs can specify executable workflows, we present an ISCO based workflow engine, acting over the information extracted from Activity Diagrams and integrated into a platform that allows a user to graphically model and manage workflows within a web environment. The server, built with ISCO, assures the data persistence and the execution of the workflows. Communication with the server is performed through AJAJ.

Resumo

Modelação de Workflows usando UML, Ferramentas Declarativas e WEB2.0

A tradução manual de diagramas UML para código computacional é tediosa e dada a erros. Muitas ferramentas CASE permitem gerar código computacional a partir de Diagramas de Classe, mas menos, se algumas, permitem a transformação de Diagramas de Actividade (DAs) em definições executáveis de workflows.
Este projecto pretende:

- Traduzir DAs UML para a linguagem de programação ISCO;
- Construir um sistema de execução que corra as traduções;
- Construir uma ferramenta gráfica para criação, edição e visualização de workflows e instâncias.

Defendendo que os DAs UML são capazes de especificar workflows executáveis, apresentamos um motor de execução de workflows baseado em ISCO, agindo sobre a informação recolhida de DAs e integrado numa plataforma que permite a modelação gráfica e gestão de workflows dentro de um ambiente web. O servidor, construído em ISCO, assegura a persistência dos dados e execução dos workflows. A comunicação entre cliente e servidor é feita com AJAJ.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The use of an UML [1] modeling tool as a means of producing programming code is a pursued objective, but still afar, due to UML not being a graphical programing language but rather a graphical modeling language [2]. The gap between modeling diagrams and producing self-sufficient functional code is clear when attending to the fact that many of the UML modeling tools only have XML (or XMI [3]) outputs, whilst those which can actually generate real programming code, like Java or C++ (ex: Rational Rose [4]), can only do so for a restricted subset of diagrams, such as class and component diagrams.

As such, the initial motivation of our work was to produce ready-to-execute ISCO [5] code, originating from UML diagrams, namely, Activity Diagrams (AD) and Class Diagrams. For the first, the code would be loaded to an execution system which would execute the given model, allowing state changes and action executions. For the second case, the output would be capable of generating a database, resulting from the created ISCO classes. The investigation to pursue these goals led to a rather large evolution of the project's scope, where a part of the afore mentioned goal is a feature of the purposed application prototype, while the second part was put aside, as discussed in section 3.2.

This way, our purpose is to establish a tool for modeling, managing and executing workflows, derived from UML Activity Diagram models, on a web based environment and with an ISCO back-end, while still allowing the source of the UML to be modeled with an external UML modeling tool.

1.2 Objectives

The goal of our project is to present a general purpose workflow engine, fully implemented in the ISCO language, that solely runs on UML Activity Diagrams. Another goal of the project is to present the common workflow modeler/user with a tool for on-line graphical workflow modeling, execution and management, using, willingly, no more than the regular browser.

As a direct result of the initial motivations described in section 1.1 (the production of ISCO code), the use of the ISCO language to build the application's back-end, allows for deeper exploration on information treatment as well as performing all sorts of tests, due to ISCO's declarative paradigm.

These tests include not only tests on the data resulting from workflow executions, but also over the workflow structure itself, where, for instance, the user might be given feedback from the application upon a structural modeling error.

The work described in this thesis was partially described in a conference article [6], in which the aspects of the graphical user interface and underlying support structures were mostly discussed.

1.3 Thesis Organization

The remainder of this thesis is organized as follows: in chapter 2 we span over the several technological areas we use in the project and try to give the reader the necessary background, so that the rest of this document and project can be better understood. Chapter 3 describes the phased evolution of the project, in a chronological fashion, presenting the choices and decisions that caused the project to move away from the initial purpose to its final and wider concept.

In chapter 4 we present and describe the system's data model, as well as some relevant aspects of internal information representation.

Chapter 5 focuses on the details of the graphical component and user interaction, that is, the web-client. We describe the overall structure of the client, listing each file and respective objectives. We then move on to cover how communications take place between the web-client the server, and present a few screenshots of the actual web-client, as well as a brief description of how to move inside the client.

Chapter 6 presents a case study, where a workflow is created, including activities, transitions and events, and then executed, all using our application. We also present a tabular view of our application's characteristics, compared to three other applications.

In chapter 7 we draw some conclusions about how and what objectives where completed, and outline possible future work.

Chapter 2

Technology Overview

In this chapter we go through the main technologies used in our work.

We start with a brief overview on workflows in section 2.1 and proceed to workflow patterns and some of their relations with and representation in UML AD in section 2.2, where we roughly compare the BPMN [7], YAWL [8] and UML AD representation of each flow pattern. We then go over a few basic aspects of ISCO Prolog, in section 2.3, followed by Apache and PHP in section 2.4. We also discuss some aspects of Ajax [9] and the toolkit with which the web application was developed - the Dojo Toolkit [10]- in section 2.5. Finally we go through some technical aspects of the SOAP [11] [12] web service protocol in section 2.6.

2.1 Workflows

A workflow is a process schematic that represents how to get get something done, through a set of steps or actions. The objective of the process can be related with "... any form of physical transformation, service provision or information processing" [13]. Despite all these options, we are only concerned about Business Process' related workflows, that is, management processes [1],

[1]Management processes are the processes that govern the operation of a system

operational processes [2] and supporting processes [3].

2.1.1 Workflow components

Workflows are usually graphically represented using formal or informal flow diagramming techniques, depicting directed *flows* (transitions) between *nodes*. Transitions and nodes are thus the main elements of workflows, though there are several types of nodes - its number depending on the chosen representation - and typically two types of transitions, depending on whether it is conditioned. Other components can be used which can help the visual understanding of the workflow, such as swimlanes, notes, signals, boxes, etc.

Tables 2.1 2.2 2.3 show the most usual graphical components found in UML Activity Diagrams [4].

Node Name	**Notation**	**Description**
Initial	●	An initial node is a control node at which the flow starts when the activity is invoked.
Activity Final	◉	An activity final node is a final node that stops all flows in an activity.
Flow Final	⊗	A flow final node is a final node that terminates a flow.

Continued on next page

Table 2.1: Graphical notation of nodes

[2] Operational Processes create the primary value stream and are part of the core business. Typical operational processes are manufacturing, purchasing, sales, etc

[3] Supporting Processes support the core processes. Examples are Accounting, IT- support, etc

[4] Tables, figures and descriptions adapted from [14]

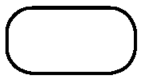
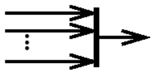
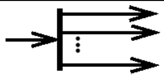
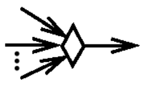
Node Name	**Notation**	**Description**
Action		An action represents a single step within an activity, that is, one that is not further decomposed within the activity. An activity represents a behavior that is composed of individual elements that are actions. Note, however, that a call behavior action may reference an activity definition, in which case the execution of the call action involves the execution of the referenced activity and its actions (similarly for all the invocation actions). An action is therefore simple from the point of view of the activity containing it, but may be complex in its effect and not be atomic.
Join		A *join node* is a control node that synchronizes multiple flows.
Fork		A fork node is a control node that splits a flow into multiple concurrent flows.
Merge		A *merge node* is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows.
		Continued on next page

Table 2.1: Graphical notation of nodes

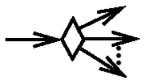
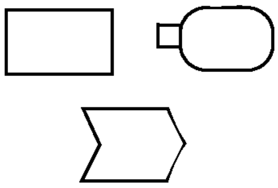
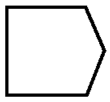
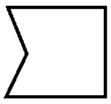
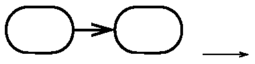
Node Name	**Notation**	**Description**
Decision		A merge node is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows.
Object		An object node is an abstract activity node that is part of defining object flow in an activity.
Send Signal		SendSignalAction is an action that creates a signal instance from its inputs, and transmits it to the target object, where it may cause the firing of a state machine transition or the execution of an activity. The argument values are available to the execution of associated behaviors.
Receive Signal		ReceiveSignalAction is an accept event action representing the receipt of a synchronous call request.

Table 2.1: Graphical notation of nodes

Flow Name	**Notation**	**Description**
Control Flow		A control flow is an edge that starts an activity node after the previous one is finished.

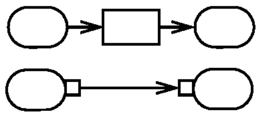
Object Flow		An object flow is an activity edge that can have objects or data passing along it.

Table 2.2: Graphical notations of Flow Shapes

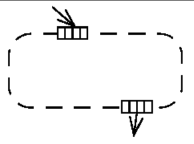
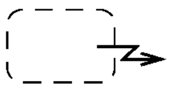
Region	**Notation**	**Description**
Expansion		An expansion region is a structured activity region that executes multiple times corresponding to elements of an input collection.
Interruptible		An interruptible activity region is an activity group that supports termination of tokens flowing in the portions of an activity.

Table 2.3: Graphical notation of regions

2.2 Control patterns in ADs

Workflows consist of a number of components which set a course (flow) for information from a point to another, performing a set of actions along the way. These components can be arranged in groups, each with a generic function, called **flow control patterns** (or simply control patterns). Workflows and their flow control patterns, have a loose way of being designed when compared with UML modeling. For instance, an action state might have any number of outgoing transitions, while in formal UML it will have exactly one.

There are roughly 21 different workflow patterns [15], which can be organized in 6 categories: basic control patterns; advanced branching and synchronization patterns; structural patterns; multiple instance patterns; state based patterns and; cancellation patterns.

2.2.1 Basic Control patterns

The first 5 patterns define the basic aspects of process control. These patterns are:

Sequence

The *Sequence* pattern is defined by an ordered series of activities, belonging to the same process, where each activity is executed after the completion of the previous. This pattern may also be called *sequential routing* or *serial routing*.

As shown in figures 2.1 and 2.2, UML AD and BPMN respectively, there are no major differences.

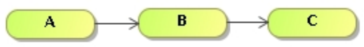


Fig. 2.1: Sequence (AD)

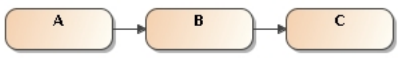


Fig. 2.2: Sequence (BPMN)

In YAWL, however, the action states (called tasks), are not represented by the usual rounded rectangles, but rather by squares.
The *Sequence* pattern can be represented in two ways, as depicted in figure 2.3, where two activities are connected by an arrow, having, or not, a circle between them - the circle represents a condition.

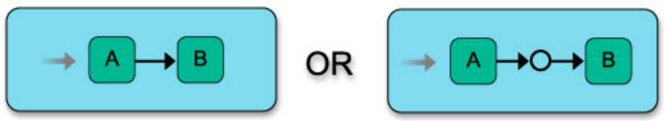


Fig. 2.3: Sequence (YAWL)

Parallel Split

A *Parallel Split* pattern is the mechanism through which a control thread splits in two other threads, which will execute concurrently and in an unordered fashion or even in simultaneous. This pattern is also called *AND-split*, *parallel routing*, *parallel split* or simply *fork*.

Although there are three different variants for creating a *Parallel Split* mechanism in BPMN, as depicted on the figure 2.4, the first option represents the usual way, with two (or more) outgoing connections directly from one action state, each to another action.

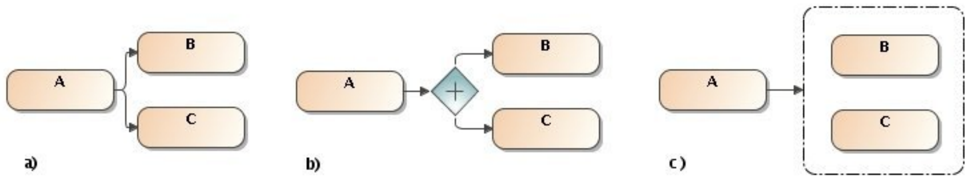


Fig. 2.4: Versions of parallel split (BPMN)

In YAWL, there's also an extra shape involved for this pattern. As seen in figure 2.5, the activity leading to the split incorporates itself the split representation.

As for the activity diagrams, the representation is a little less simple, as there is the need to add an extra graphic shape, the *fork*, a vertical or horizontal bar, as seen in figure 2.6. This shape has one single incoming transition and how many as desired outgoing transitions, each representing one parallel control path.

Synchronization does not concern this pattern, and it may or not be

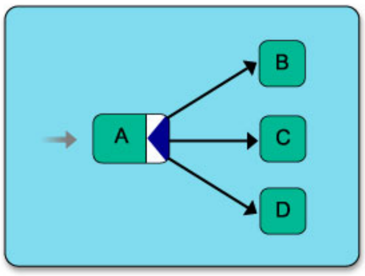


Fig. 2.5: Parallel Split (YAWL)

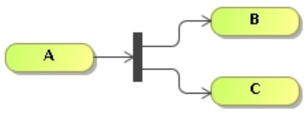


Fig. 2.6: Parallel Split (AD)

present in a workflow using this pattern.

Synchronization

A *Synchronization* pattern is used to converge two or mode parallel paths, typically ending a parallel execution. This pattern assumes that each of the execution paths are executed only once.

Just as the previous pattern, the BPMN representation of a *Synchronization* can be done in more than one way, as shown in figure 2.7.

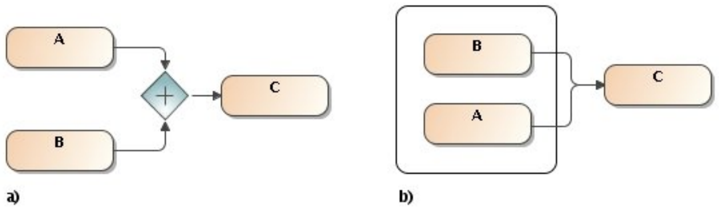


Fig. 2.7: Synchronization (BPMN)

Thus, in BPMN, we can have either no explicit shape for this pattern, given we are working with a “parallel box”, 2.7b, or can use the exactly same diamond shape with a plus sign, as in the *Parallel Split*, but with different semantics, as shown on 2.7a. In fact, this particular shape can represent both node types simultaneously, with multiple incoming and outgoing transitions.

In YAWL, similarly to what happens in the *Parallel Split* pattern, the shape indicating the synchronization of two or more flows is also appended to one activity, as it can be seen in the figure 2.8.

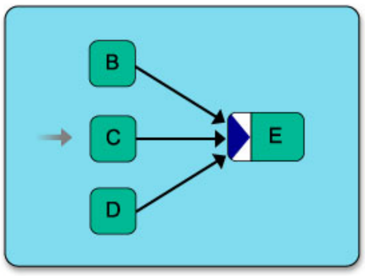


Fig. 2.8: Synchronization (YAWL)

The AD counterpart is stricter than BPMN, having a designated node type for this function, namely, the *join* node. This node looks just as a *fork* node, with the difference that we now have multiple incoming and exactly one outgoing transitions, as illustrated in figure 2.9.

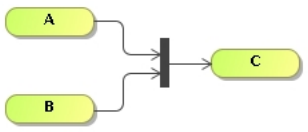


Fig. 2.9: Synchronization (AD)

The UML2.0 revision states that one horizontal or vertical bar can behave like the mentioned BPMN plus signed diamond, having multiple incoming and outgoing transitions. Despite this fact, we, in our application, use the two different types (*split* and *fork*), due mainly to engine implementation reasons and also because the use of both functions simultaneously (incoming and outgoing flows) is not very common. We do allow more than one incoming transition into a *fork*.

Exclusive Choice

An *Exclusive Choice* is defined as being the place in the process where the flow path is dependent of a choice, where only one of the possible paths is followed.

The typical representation is by means of a diamond shaped node, called *split*, as for both BPMN (which adds an *X* inside as seen in figure 2.10) and UML (figure 2.11).

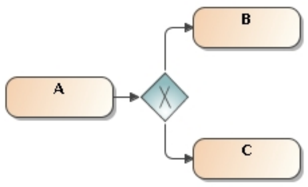


Fig. 2.10: Exclusive Choice (BPMN)

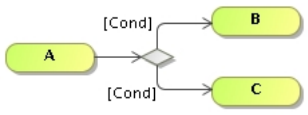


Fig. 2.11: Exclusive Choice (AD)

In YAWL, the representation is very similar to that of a *Parallel Split*, with the difference of the arrow of the box appended to the activity being directed in the opposite direction, as shown in figure 2.12.

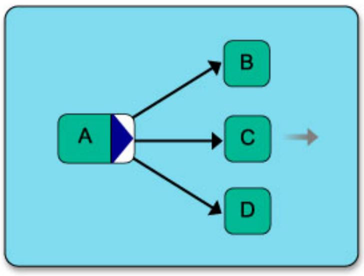


Fig. 2.12: Exclusive Choice (YAWL)

As with some other workflow engines (MQSeries Workflow [16], Verve [17]) the exclusiveness of the *Split* node is left for the user to simulate in the outgoing conditions. In YAWL, when multiple conditions evaluate to true, there is a preference, set at design time, in each transition.

Simple Merge

A *Simple Merge* pattern is defined by a point in the process where two or more flow paths are merged together, without any kind synchronization.

The BPMN has, once more, two different ways of representing this pattern, with different behaviors (figure 2.13). One way is done by uncontrolled flow, where no control node is present between the two simple activity nodes. The other uses a diamond with an `X` sign to represent an *exclusive gateway*. Since for this last scenario only one token is expected, as the multiple flows are usually modeled originating in an *exclusive choice* pattern, the behavior of the merging is straight forward. If, on the other hand, there are more than one active flows that lead to the *exclusive gateway*, only one (the first) will be allowed to proceed. In this scenario this pattern acts like a *discriminator*, seen in section 2.2.2

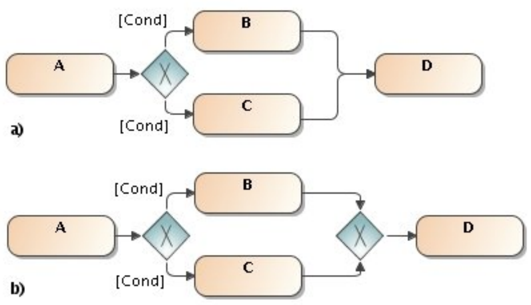


Fig. 2.13: Simple merge options (BPMN)

In YAWL, the representation is very similar to that of a *Synchronization*. The difference is, as with the previous pattern, in a reversed arrow, as shown in figure 2.14. The *Simple Merge* and *Multiple Merge* are both represented equally.

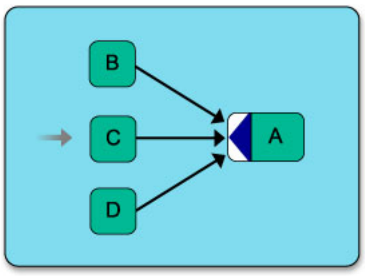


Fig. 2.14: Simple/Multiple Merge (YAWL)

The UML AD representation uses a diamond shape similar to the *split* node, called a *merge* node. Technically, the *merge* node can be omitted from the control flow, as a *simple activity* accepts multiple incoming transitions, but this is not considered the best practice. Figure 2.15 illustrates the *Simple Merge* pattern in UML AD notation.

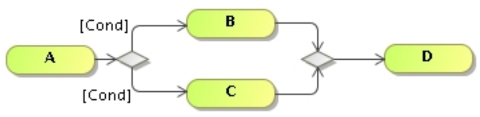


Fig. 2.15: Simple Merge (AD)

We would like to point out that in [15] the *exclusive gateway* is represented as a simple *gateway*, that is, with no X inside.

It's also worth to mention that [15] states that for the scenario depicted in figure 2.13a, that is, when an *exclusive gateway* is used, the representation will be equivalent to that of UML DA's. Though this is visually true, the behavior is different since, as stated in [14], all tokens that reach a *merge* node will be carried forward.

2.2.2 Advanced Branching and Synchronization patterns

The next five patterns describe more complex ways of branching and merging the process flows.

Multiple Choice

The *Multiple Choice* pattern differs from the *Exclusive Choice* pattern by allowing the choice of more than one flow paths, depending on how many conditions are validated as true.

Once again, the BPMN allows more than one representations, either by adding small diamonds to each outgoing transition or by adding a circular marker to a single diamond, seen in figure 2.16.

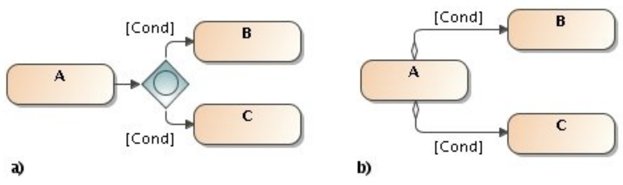


Fig. 2.16: Multiple Choice (BPMN)

In YAWL, the representation of this pattern is again very similar to a *Parallel Split* or *Exclusive Choice*, but this time, with a diamond shape instead of the directed arrows, as shown in figure 2.17.

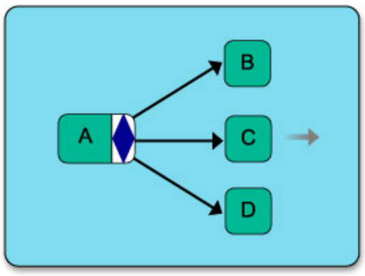


Fig. 2.17: Multiple Choice (YAWL)

The UML AD representation uses *fork* nodes, combined with condition guards on the outgoing transitions [5], as shown in figure 2.18.

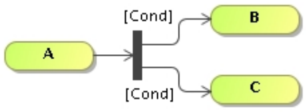


Fig. 2.18: Multiple Choice (AD)

Multiple Merge

The *Multiple Merge* is defined as a place in the process where two or more flow paths are merged together as one.

For the BPMN, no special shape is used, being that this pattern is simply represented by multiple transitions entering an activity node. This pattern also involves an uncontrolled flow situation, which is the same as depicted in figure 2.4a. Therefore, for each token that reaches activity D in figure 2.19 an instance is created.

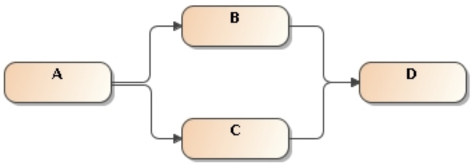


Fig. 2.19: Multiple Merge (BPMN)

[5]For the time-being, conditional transitions are only supported as output from decision nodes (*split* nodes), in our application.

Regarding UML AD and YAWL representations, there are no differences between this and the *Simple Merge* pattern, as supported in [18], except that there will be multiple parallel flows entering the *merge* node.

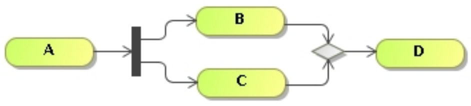


Fig. 2.20: Multiple Merge (AD)

Synchronizing Merge

A *Synchronizing Merge* pattern is defined as the place where two or more *active* flow paths are merged together. This means that the *Synchronizing Merge* pattern must know how many tokens were generated earlier, then synchronize those tokens, but not wait for any other tokens. If there is only one active path that leads to the merge node, no synchronization will happen.

Although in [15] it is mentioned that the BPMN uses a *multi choice* node and an *inclusive merge* to achieve this pattern, as shown in figure 2.21, it is not clear how the knowledge of how many active paths actually reach the *inclusive merge.*

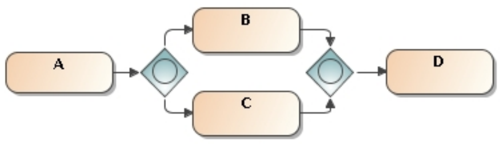


Fig. 2.21: Synchronizing Merge (BPMN)

As for YAWL, we see in [18] that it uses a special symbol, *or-join* (diamond shape on the side-box), which "is enabled if and only if an incoming branch has signaled completion and from the current state it is not possible to reach a state where another incoming branch signals completion". A simple scenario is shown in figure 2.22a and a more complicated one in 2.22b, where the *or-join*, appended to task D, is only activated depending on the exclusive choice between tasks E and F. If task E is chosen, D is activated as there are no more possible transitions to task D.

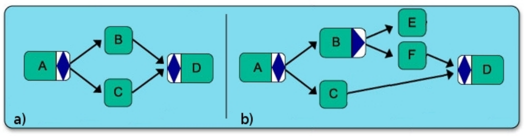


Fig. 2.22: Synchronizing Merge, a) simple scenario and b) complex scenario (YAWL)

[15] presents the UML AD approach for this pattern to be a combination of a *join* node and a condition expliciting how many needed flows must there be for the *join*. Although these flows are originated by a *fork* node, this node obeys to the definition of the *Multiple Choice* pattern (section 2.2.2), where its various outgoing transitions are guarded with conditions. This is shown in figure 2.23.

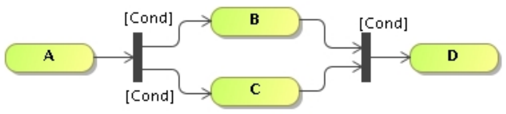


Fig. 2.23: Synchronizing Merge (AD)

[18] identifies several problems in the previous representation and further states that “no direct support is provided for this pattern” in UML ADs. One of the problems pointed out concerns the condition in the *Join* node and how it’s “not specified and it is not clear how it could be determined how many tokens to expect”. Regarding this aspect, our opinion is that the question of how many tokens to expect should be an issue to resolve on the engine rather than on the representation.

A theoretic implementation proposal for this pattern is presented in chapter 7.2.2.

Discriminator

The *Discriminator* pattern can be defined as the place where several flow paths are merged together without synchronization, and only one token (the first to reach the *Exclusive Gateway* or *Join* node) is carried forward, without waiting for any other token. All other eventual tokens are ignored.

The BPMN uses a simple *Exclusive Gateway* (diamond shape with an `X` inside), with multiple incoming transitions, to represent this pattern.

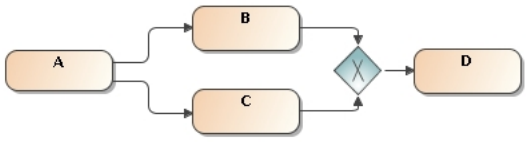


Fig. 2.24: Discriminator (BPMN)

YAWL makes use of a *cancellation region* (a dotted area) along with a *multiple merge* pattern. The concept in YAWL is a bit different, as the cancellation region actually cancels any other executing incoming branches following the first to complete. This representation can be seen in figure 2.25.

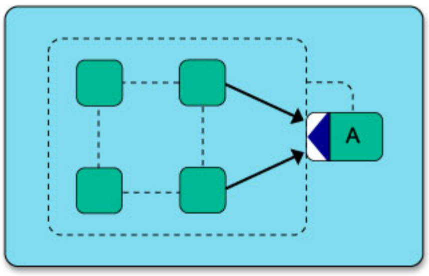


Fig. 2.25: Discriminator (YAWL)

According to [15] the UML AD notation uses a *join* node with a condition set to block all but the first token to reach the *join*, as shown on figure 2.26.

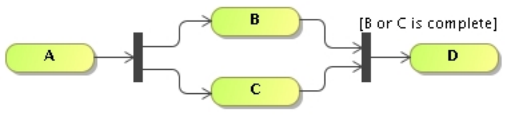


Fig. 2.26: Discriminator (AD)

[18] also presents this pattern as a specialization of the *N-out-of-M Join* pattern (in section 2.2.2) with the corresponding condition set to allow only one flow.

N out of M Join

The *N out of M Join* pattern can be defined exactly as the previous pattern, with the difference that the *join* node will wait for N tokens before allowing the flow to go on and block/ignore any posterior eventual tokens.

BPMN uses a *complex gateway* (diamond with an asterisk), included in the notation for this particular situation, as seen in figure 2.27.

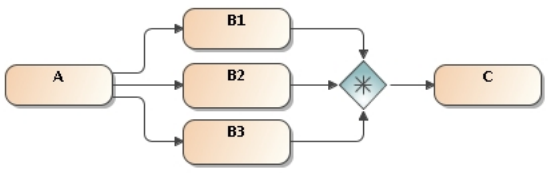


Fig. 2.27: N-out-of-M Join (BPMN)

YAWL has as particular and explicit way for representing this pattern, as shown in figure 2.28, where the *N* and *M* values are visible in the design.

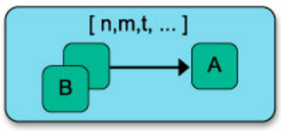


Fig. 2.28: N-out-of-M Join (YAWL)

[18] presents an UML AD representation, shown in figure 2.29 which makes use of the *InterruptibleActivityRegion* ([14] on p.377, and table 2.3) together with *weights* ([14]on p.327).

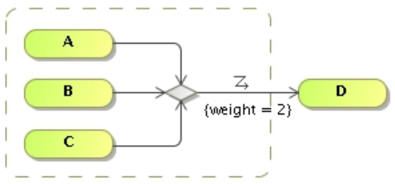


Fig. 2.29: N-Out-Of-M Join (AD)

While [2] presents a complex representation of this pattern, we take full advantage of our implementation properties of the *Join* activity, which was developed with an internal counter stating how many incoming flows must have been completed for the join to be activated. This counter is technically equivalent to the condition referred to in [15].

2.2.3 Structural Patterns

The next two patterns describe the way of achieving loops and multiple flow termination points.

Arbitrary Cycles

An *Arbitrary Cycle* is defined as the section in the workflow process where one or more activities can be done repeatedly.

In BPMN, UML AD and YAWL this can be achieved by connecting downstream activities with upstream activities, although the UML AD and YAWL notations will use a few more shapes than BPMN to achieve the same results, as shown in figures 2.30, 2.31 and 2.32.

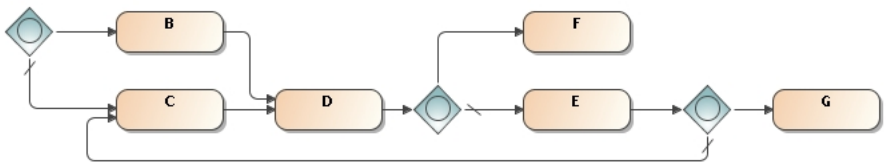


Fig. 2.30: Arbitrary Cycle (BPMN)

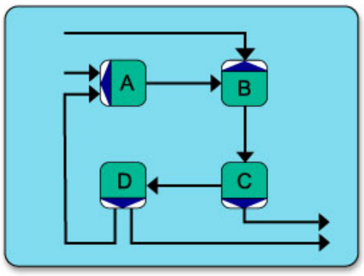


Fig. 2.31: Arbitrary Cycle (YAWL)

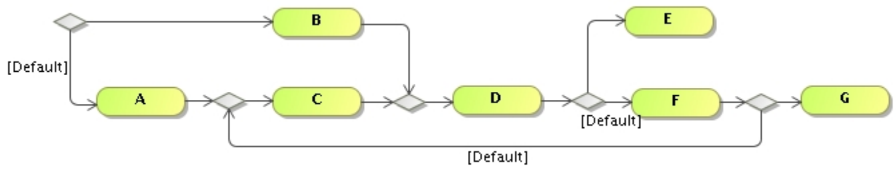


Fig. 2.32: Arbitrary Cycle (AD)

Implicit Termination

An *Implicit Termination* allows that only a particular flow path be concluded without having any interference with any other possible parallel flow paths.

In BPMN, an *End Event* (an empty circle with thick border) signals the end of the flow path, as shown in figure 2.33. The several (7) variations of the *End Event* node allows the indication of a particular result when ending the flow. The *implicit termination* pattern is valid for all of them, except the *Termination End Event*.

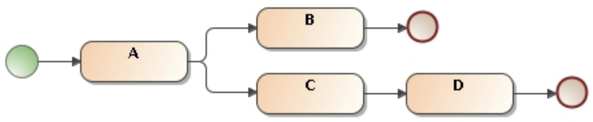


Fig. 2.33: Implicit Termination (BPMN)

UML AD notation uses a *Flow Final* node (a crossed circle) to represent the flow termination.

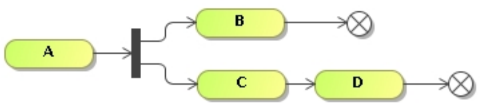


Fig. 2.34: Implicit Termination (AD)

As for YAWL, it "deliberately does not support implicit termination in order to force workflow designers to think carefully about workflow termination" ([18]).

2.2.4 Patterns Involving Multiple Instances

The next four patterns describe the situations where there can be multiple instances (MI) of an activity active simultaneously, in the same workflow process. YAWL directly supports Multiple Instance patterns through the construct seen earlier in figure 2.28.

MI with A Priory Design Time Knowledge

This pattern describes how an activity can be instantiated a *known* number of times, and in parallel. At design time, the modeler knows how many instances of the activity are desired and hardcodes it in the representation.

The definition of the pattern being vague, two variations may arise: a) where the instances and tokens are to be synchronized before continuing and; b) where the process can continue independently for each token. For a), the *MI requiring Synchronization* pattern is advised, and a variation of the *MI with no A Priori Knowledge* pattern is advised for b) [15].

The BPMN representation of this patterns is achieved by setting an attribute, which will result in the two vertical parallel lines appearing on the activity to have multiple instances.

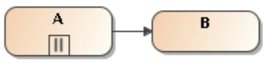


Fig. 2.35: MI with A Priori Design Time Knowledge (BPMN)

For YAWL, [18] states that this pattern is supported by “replicating the activity involved as many times as required”.

YAWL provides direct support for the MI patterns by means of “four attributes: the minimum number of instances to be created; the maximum number; a threshold for continuation (where the semantics is that if all created instances have completed or the threshold has been reached the multiple instance task can complete); and an attribute with the possible values *static* and *dynamic* indicating whether or not it is possible to create new instances when a multiple instance task has been started".

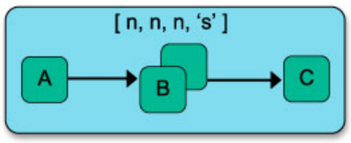


Fig. 2.36: MI with A Priori Design Time Knowledge (YAWL)

According to [15], the representation of this pattern in UML ADs is through the use of *Expansion Regions* ([14] p.365, and in table 2.3), as shown in figure 2.37.

Although in [18] it is said that this pattern is possible in UML AD by "replicating the activity involved as many times as necessary", a graphical representation is not presented.

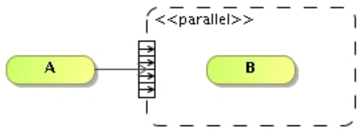


Fig. 2.37: MI with A Priori Design Time Knowledge (AD)

MI with A Priory Runtime Knowledge

This pattern is similar with the *MI with a priory design time knowledge*, the difference being that "the number of copies is not known until the process is being performed and cannot be set ahead of time" [15]. Additionally, this pattern allows that the instances are in parallel as well as in sequence.

The BPMN enables this pattern by setting attributes such as *LoopType*, *MI_InstanceGeneration* and *LoopCondition* to a specific combination of settings. Graphically, there will be a loop icon on the repeated activity, as seen in figure 2.38. [15] mentions yet another representation for this pattern "by using an exclusive Gateway Decision after the activity and looping back to merge with the same activity (...) for each copy of the activity."

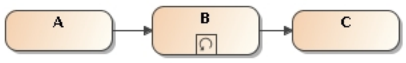


Fig. 2.38: MI with a priori runtime knowledge (BPMN)

Similarly to BPMN, in UML ADs notation this pattern can be represented "by using a decision node after the activity and looping back to merge with the same activity (...) for each copy of the activity" [15].

UML ADs can also use the *Expansion Region*, similarly to the previous pattern, taking advantage of its attributes to achieve the desired behavior. In this case the *Mode* attribute is set to Iterative rather than Parallel, as shown in figure 2.39.

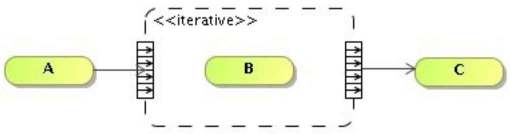


Fig. 2.39: MI with a priori runtime knowledge (AD)

MI with no a priory knowledge

The *MI with no a priory knowledge* pattern differs from the previous patterns in that the number of instances to be created is influenced by the instances themselves. This adds more complexity to the representations, though both BPMN and UML ADs notation supports this pattern, and do so with no major differences.

For BPMN, a combination of components such as seen in figure 2.40 is used.

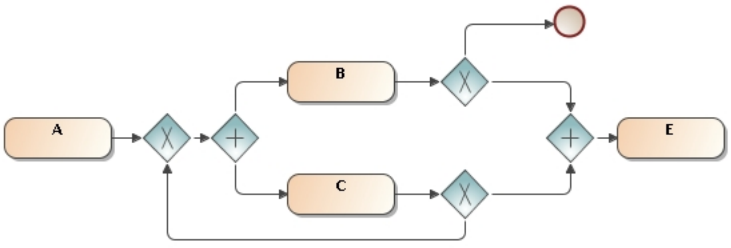


Fig. 2.40: MI with no A Priori Knowledge (BPMN)

UML AD notation uses a "combination of a decision node, merge node, fork node and activities" [15], as seen in figure 2.41. Activities B and C are started in parallel by the fork node, but a decision node immediately follows activity C. If any another copy of B is needed, the control flow will then connect upstream to a merge node. The merge node is required to avoid the synchronization behavior of the fork node (which can also be a join node).

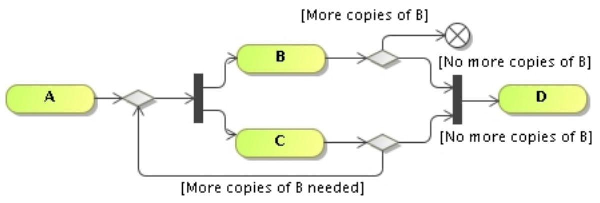


Fig. 2.41: MI with no A Priori Knowledge

MI requiring Synchronization

This pattern is similar to the *MI with A Priori Runtime Knowledge* pattern, with the difference that all the instances of the repeated activity be complete before the process continues. The instances must also be performed in parallel.

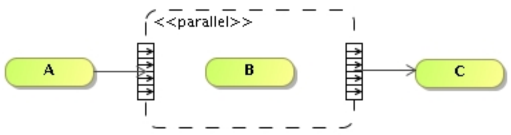


Fig. 2.42: Mi requiring Synchronization

For BPMN, the representation of this pattern is equivalent to a *MI with A Priori Design Time Knowledge*, the differences being in the internal variable that are set to specify the desired behavior. Visually, the pattern is as depicted in figure 2.35.

2.2.5 State-based patterns

The next three patterns cover the way a business process is sometimes affected by factors outside the direct control of the process engine [15] which may be caused by, among other things, the unavailability of human or material resources [2]. There is consequently an interval between the moment when an activity is enabled and when its execution starts.

Deferred Choice

A *Deferred Choice* pattern is used when there is the need to choose among several control flows, though the choice "is not based on data that is available at the moment the execution reaches the deferred choice, but is rather determined by an event (e.g. an application user selecting a task from the work list, or a message being received by the process execution engine)".

In BPMN this pattern is represented as shown in figure 2.43, using a *Gateway* (diamond with a star inside), where the control flow will wait until the specific event occurs (usually a message - represented by the circles with an envelope inside). Other events can be used. Only one control flow will be chosen, as the first event to occur will trigger the correspondent control flow, ignoring the other flows.

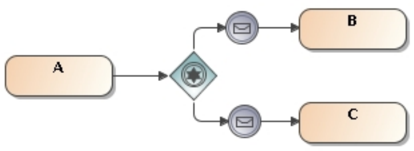


Fig. 2.43: Deferred Choice (BPMN)

[18] states that, since YAWL is based on Petri nets, it directly supports the *deferred choice* pattern, which is shown in figure 2.44. A *place* (condition represented by the circle) performs the decision and, at runtime, the alternative

that is chosen consumes the token thus disabling the other alternatives [18].

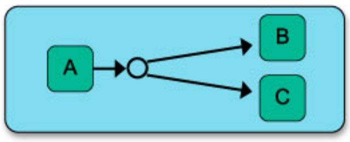


Fig. 2.44: Deferred Choice (YAWL)

In UML AD notation, the pattern can be represented as shown in figure 2.45a, which makes use of *Signal Events* following a *Fork*. The *Signal Events* are also encircled by an *Interruptible Region* line, which will disable all the other *signal events* after the first is activated, thus achieving the desired deferring choice behavior.

[2] presents another representation of this pattern, expressing it as "a normal state which waits for an event from the environment, and chooses one of its outgoing branches accordingly". Figure 2.45b depicts this representation.

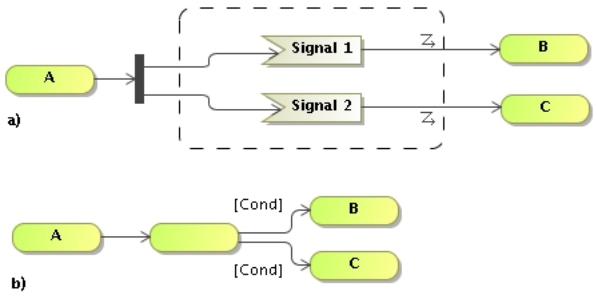


Fig. 2.45: Deferred Choice (AD)

Interleaved Parallel Routing

Though using the word "parallel", this pattern is defined by a set of activities that must be executed sequentially, exactly once and with no particular

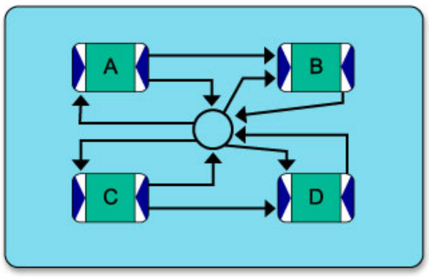


Fig. 2.47: Interleaved Parallel Routing (YAWL)

order. The sequential performance of the activities is often due to the requirement that the activities share or update the same resources [15].

In BPMN, the *Ad-Hoc Process* (a *Sub-Process* box with a tilde on the bottom) is used, semantically meaning a collection of activities that can be performed in any order. It also requires one attribute to be configured so that the sequential order is accomplished, as well as a modeler-defined condition that must be satisfied for the sub process to complete. This representation is shown in figure 2.46.

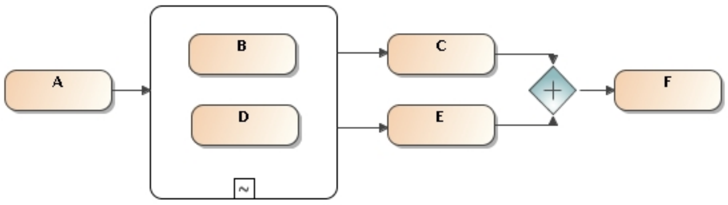


Fig. 2.46: Interleaved Parallel Routing (BPMN)

As for YAWL, the solution is based in a *mutex place* (*mut*ual *ex*clusion), also represented by a circle. This is shown in figure 2.47. None of the tasks A, B, C, or D are executed in parallel. In addition, the execution of task B has to await completion of the execution of task A, and the execution of task D has to await completion of the execution of task C (from [8]).

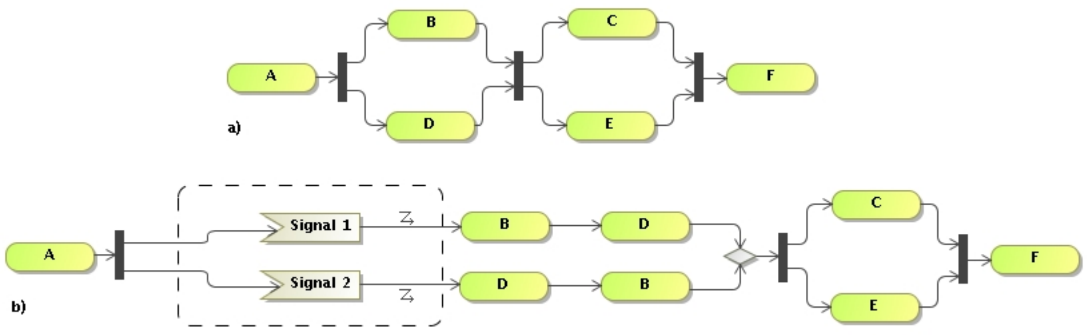


Fig. 2.48: Interleaved Parallel Routing (AD)

In UML AD notation there is no direct support for this pattern, as there is no concept of an Ad-Hoc process, but workarounds can be used, and [15] proposes two.

The first and simplest solution is to place the activities (B and D) between a *Fork* and *Join* pair. As is, this would allow the activities to be executed in parallel, so constraints must be added (one to each activity) so the use of the same resource is forced upon them, semantically meaning that they cannot execute at the same time. This is shown in figure 2.48a.

The second representation, more explicit, involves the use of the *Deferred Choice* pattern as well as duplicates of some activities (again B and D), as shown in figure 2.48b.

In regard to the first solution, the management of the resources is not intended to be controlled lest touched by the workflow engine in our application. This means that this pattern can be represented as expressed, leaving the resource management to be handled by the "system" that actually executes the actions.

For the second solution, and since it's represented at the expense of the *Deferred Choice* pattern, it is not supported by our application.

Milestone

A given activity can only be enabled if a certain milestone has been reached which has not yet expired. A milestone is defined as a point in the process where a given activity has finished and an activity following it has not yet started [18].

Since the representation of this pattern is not directly available for UML DA notation, the mimicking of Petri Net's *places* with signal exchanges can add complexity to the models. Adding to this is the fact that many of the workarounds assume specific semantics for UML AD [18], which may lead to interpretation errors. Also in [18], a suggestion is expressed that there is no consensus on the semantics of the more advanced representations.

Based on this, we decided not to go into any level of detail for this pattern.

2.2.6 Cancellation Patterns

The purpose of the next two patterns is to define a way that the completion of an activity causes the cancellation of another activity or group of activities.

Cancel Activity

In this pattern, there is the need to signal the cancellation of an activity and the need act (cancel an activity) based on that signal, as the completion of one of the two (or more) competing activities means the cancellation of the others.

BPMN uses *intermediate events* attached to the boundary of the activities, as shown in figure 2.49. If that event is triggered, the activity will immediately be canceled, allowing the control flow to follow its path.

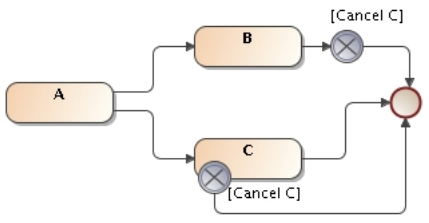


Fig. 2.49: Cancel Activity (BPMN)

YAWL uses *Cancel Regions* connected to activities, as shown in figure 2.50. When activity B is completed, the activity within its *Cancel Region*, A, is canceled.

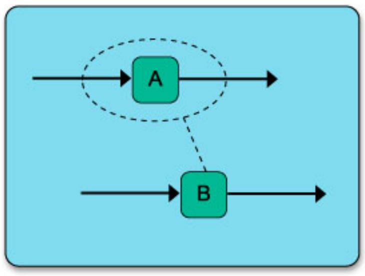


Fig. 2.50: Cancel Activity (YAWL)

In UML AD, this pattern also makes use of an *Interruptible Region*). When a signal inside the region is triggered, it will result in the cancellation of the other activities inside the region. The origin of the signal can be as shown in figure 2.51, where upon the completion of activity B the *Cancel C* trigger is fired. The signal will then cross the border of the region and cancel activity C.

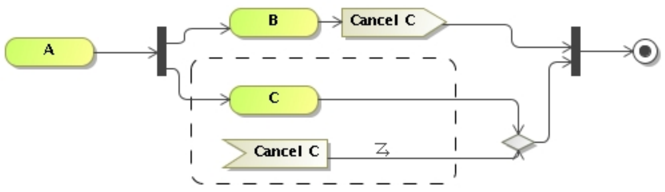


Fig. 2.51: Cancel Activity (AD)

Cancel Case

Cancel Case is actually an extended *Cancel Activity*, as its definition is the same except that the goal is to cancel a process rather than just one activity.

BPMN represents this pattern by reallocating the *intermediate event* from the border of the activity to the border of a *Sub-Process* containing the desired activities, as shown in figure 2.52a. Another scenario (in [15], though the distinction is not made clear) is when a modeler wants the cancellation of the entire process. In this case the *Terminate End Event* should be used, as shown in figure 2.52b.

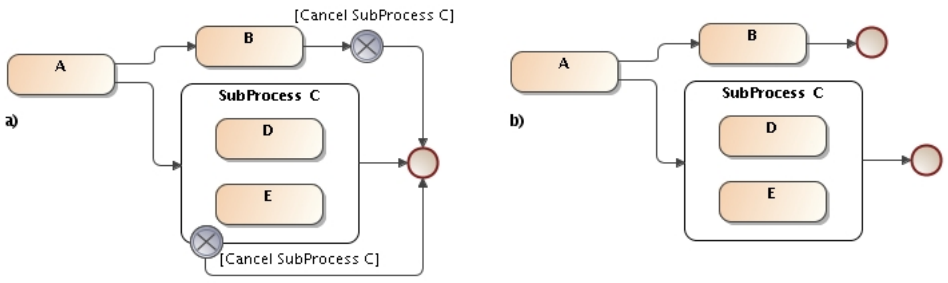


Fig. 2.52: Cancel Case (BPMN versions)

In YAWL, the representation is equivalent to the *Cancel Activity*, as the *Cancel Region* may encircle "... a single task, a whole case, and anything in between" [18].

UML AD notation for this pattern is also very similar to that of a *Cancel Activity*, but where, in the last pattern, we had a region surrounding an activity, we now have that same region surrounding a *complex activity*, as shown in figure 2.53a. Again, if the modeler wants to cancel the whole process, an *End* node should be used, as shown in figure 2.53b.

2.3 ISCO Prolog

ISCO (*I*nformation *S*ystems *CO*nstruction) is a mediator language in that an ISCO program may transparently access data from several distinct sources

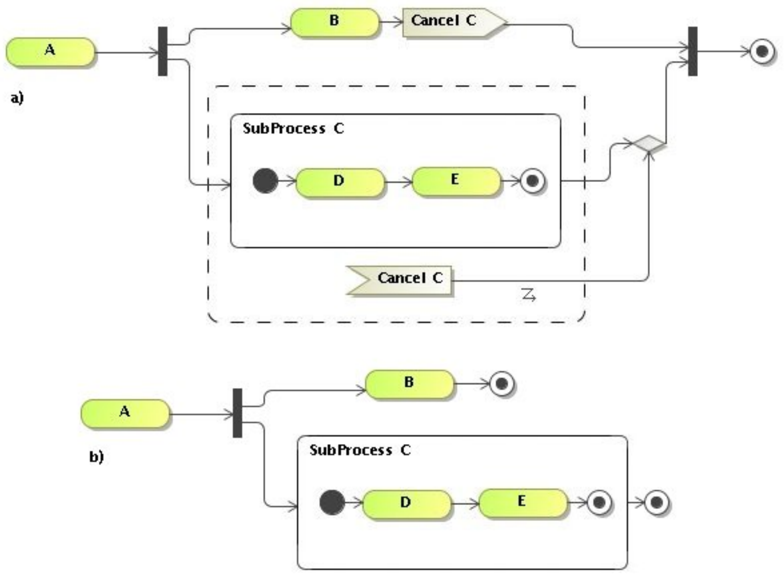


Fig. 2.53: Cancel Case (AD versions)

in a uniform way, that is, as regular Prolog predicates. Some relevant advantages ISCO holds over competing approaches are its ability to concurrently interface to several legacy systems, its high performance by virtue of being derived from GNU-Prolog and its simplicity [19].

ISCO also benefits from the integration with Contextual (Constraint) Logic Programming [20] (CxLP), a language that extends logical programming with mechanisms for modularity.

An ISCO class, consisting in a data structure definition equivalent to that of a database table, when compiled, triggers a set of changes in the back-end database. In practice, the database tables are created as well as the mechanism to access those tables. This mechanism consists, of course, in predicates.

An example of these class definitions is illustrated in figure 2.54.

```
class teacher.
    name:       text.
    department: text.
    degree:     text.
```

Fig. 2.54: ISCO class example.

2.4 Apache HTTP Server and PHP

Apache HTTP Server [21] is the most popular web server on the Internet since 1996. PHP [22] is a widely-used general-purpose scripting language that is especially suited for Web development and can be embedded into HTML.

Together, the Apache HTTP Server and PHP make a widespread and solidly installed base on which it is simple and safe to host web applications.

2.5 AJAJ and the Dojo Toolkit

From the point of view where one of the means to reach our goals involves having an accentuated interaction with the server, the use of asynchronous communication was immediately appealing.

To perform these asynchronous communications there are two disputing technologies: Ajax ([9] [23]) and the relatively new AJAJ (Asynchronous Javascript and JSON), where the XML text is replaced by JSON's [24] serialized objects.

Our choice went to AJAJ based on not so much the opinions scattered throughout the web, which can be contradictory ([25], [26], [27]), but on our already established use of Javascript code on the client side, which takes great advantage on the use of the JSON notation, that is, Java objects.

From between many [6] asynchronous Javascript frameworks and toolkits, only one presents all the graphical capabilities needed to draw a set of SVG geometric shapes on the browser, together with the manipulation tools, like drag-and-drop, for those same shapes. Its name: DOJO Toolkit [10]. Al-

[6]A list can be seen in *http://chandlerproject.org/Projects/AjaxLibraries*

though in heavy development, and not having, at the present time, one of the components to draw ADs (arrowheads), we believe that the wide acceptance of the DOJO Toolkit will have it escalate into having that missing piece. Aside from the arrowheads, all the geometric shapes can be manipulated to fulfill the requirements to draw ADs.

As a side note, although working with Dojo Toolkit is quite motivating and it still being the only right-tool-for-the-job, the lack of documentation can be troublesome, decreasing development productivity in great extent.

2.6 SOAP

SOAP [11] is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention [12] for representing remote procedure calls and responses. SOAP stood for Simple Object Access Protocol, though as of Version 1.2 [11] the acronym was dropped.

The reason why we use SOAP is actually very straightforward: because it is simple. Simple to read, as it's human readable; simple to use for both clients and servers.

The security context that SOAP lacks can be, if necessary, dealt with by some extensions to SOAP, like analyzed in [28].

Although the use of XML can make SOAP messages quite verbose, we are at ease with this situation for two reasons: a) most of the messages are expected to be extremely short, as they will mainly be remote procedure invocations and small responses, and b) even if the responses increase in size, corresponding to some more elaborate data structure, the natural response time of the typical people involving workflow is far slower than any SOAP performance issues.

2.7 Graphviz

Graphviz [29] (Graph Visualization Software) is a set of tools and algorithms to work, transform and visualize graphs. It uses The DOT Language [30] to draw direct or indirect graphs. The graph's information is loaded from attributed graph text files and the output is either in graph files of in graphic format, such as GIF, PNG, SVG or PostScript.

Regarding the use Graphviz in our application, and with the sole purpose of making the life of the modeler easier, a bit of work was put into having the capability of rearranging the organization of the modeled workflows in an automated but on-demand fashion.

2.8 Summary

In this chapter we have spanned through the main technologies that were put to use in our project. We have covered the basics about workflows, workflow components and their graphical notations. We have given some emphasis to the differences between the several kinds of control patterns, as well as to the differences between the representations of those same patterns using YAWL, UML AD and BPM notations. The basics of the ISCO programming language were also covered in this chapter, as was the justification of using the Apache HTTP Server, PHP, AJAJ, the Dojo Toolkit and, finally, Graphviz.

Chapter 3

Methods and procedures

In this section we describe the methods and procedures used for both decision and accomplishment of the intermediate objectives that were dealt with in this project. This section presents the chronological evolution of the project, discussing how some ideas where put aside and how many others came to life.

We start by covering some ground on the general opinions about the adequacy of UML Activity Diagrams to specify workflows, in section 3.1, which is followed by section 3.2, where we focus on the UML origins and purpose regarding the project, as well as discuss how to get the model information (workflows designs) into out application. In section 3.3 we briefly introduce the major concepts about the ISCO based data structure. Section 3.4 presents the workflow execution engine and its console interface [1]. The evolution of the core interfaces is explained in section 3.5, and special attention is given to how and why the use of Web2.0 came to mind. In section 3.6, the concept of "external event" and its use and API are presented, and in section 3.7 we discuss and exemplify the grammar used in specifying *transition guards*.

[1]By "console interface" we mean the way the user can initiate new workflow instances or advance through existing ones.

3.1 Regarding workflow specification using UML ADs

Though UML fails to capture some aspects of workflow specifications, as pointed out by [2], and as seen in section 2.2, UML is still a natural choice for representing business processes, as indicated by [31]. What the UML representation lacks can be compensated by extending it (in our application) in order to be capable of representing all the control patterns without hindering the use of this so well known notation. [31] reminds us that UML has been conceived for the communication among people and that, as a consequence, it does not have well defined operational semantics and it's not executable. Being that this execution property is what we pursue, some minor extensions are (or will be in future work) in order.

3.2 UML based editor

As already discussed, the initial goal of the project was the generation of ready-to-use ISCO code from UML diagrams. We have also already mentioned that we started with two UML diagrams - class and activity diagrams. Although this is correct, it soon came to our attention that the task of dealing with activity diagrams and transforming them into workflows, could be far more interesting and worthwhile than creating database managing systems from class diagrams. At this moment, all efforts where dedicated to the ADs and workflows.

ArgoUML [32] was the chosen UML editor in which to perform the modeling. Being open source as well as having XMI [3] output capabilities and supporting OCL (Object Constraint Language)[2]. The interface is also user-friendly, and the fact that it is implemented with Java makes it an easy-to-deploy application.

The choice of an editor that could easily output a XMI representation of the models did not happen by chance. Having the model in a XML based

[2]At this stage, OCL was thought of as to enable easier communication between the modeled specifications and the final ISCO code

representation allows the information harvest to happen comfortably, using pretty much any programming language that can deal with strings. This way, setting off from the XMI, the prime focus is to "translate" the activity diagram information into ISCO code.

3.2.1 XMI to ISCO translation

XSL is the only language whose sole purpose is to analyze and transform XML, as mentioned in [33], which means that *XSL Templates* [34] (XSLTs) should have no trouble dealing with the XMI output. Being that XSL is mainly used to transform XML in other XML, or other markup language (like HTML) we have to evaluate the chance of having to use an intermediate representation (IR) for our transformations, as advised in [35]. Early tests revealed that our developed XSLTs can output ISCO code without the need of the IR. We do bear in mind, though, that the UML components used in the tests where basic, and that complex representations, emerging from the use of other UML components, might lead to the need of an IR.

3.3 ISCO based data structure

ISCO's Object-Relational basis is the adopted mechanism in which we build all our data structure. This allows us to create the system data model (database) from an ISCO class description, as well as allowing the database to be seen as part of an object oriented declarative/deductive database, where each table is mapped to a class which, in turn, can be used as Prolog goals. This means that, by generating ISCO code from XMI, we automatically have in this code all the means to introduce the information in the system.

The data model is built contemplating two different, though connected, types of information. These types are *static information* and *dynamic information*, explained next.

Static information Also called "modeling information", which deals with the model related data such as activity and workflow names, screen coordinates and relations of activities, transitions, events and conditions.

Dynamic information Also called "execution information", which deals with all the relevant data related to workflow, activity and event instances. This information is usually the result of the execution of instances.

Several changes where made to the initial data model, as new necessities came along, resulting from the changes of scope of the application. Although no more changes in the scope of the application are expected, we believe that with the incorporation of more capabilities new or different representations of data may be necessary.

For a detailed insight of the complete data structure of the project, the reader is referred to chapter 4, and to the appendix (page 103) for the entity-relation model and data properties.

3.4 Execution engine

One of the most important components of this project is the workflow engine. This engine is entirely implemented using the ISCO language, taking full advantage of the knowledge available in the data structure, previously mentioned. This engine aims for the instantiation of the workflow models and the respective activities, allowing the flow path to be followed and issuing activity events where due, that is, when such was modeled on the respective AD.

Console API

It is possible to control the workflow engine manually from a text console, by calling the ISCO executable file, and using the available set of predicates.

Those predicates are:

`start/2` `start(+Workflow, ?Id)`	This predicate will create a new instance of workflow *Workflow* and return the new instance's *Id*. The new instance will be an object of the form: `execution(Workflow, Id, 0, _, stub(X)-X)`
`step/2` `step(+Workflow, +Instance)`	This predicate causes an instance to step forward, that is, it will check for any completed activities and, if possible, advance to the next activity.

3.5 Interfacing with the engine

This section describes the main problems, resolutions and consequences of deciding what would be the best interface with the application's core.

3.5.1 First interaction-model

The general idea of interacting with the execution engine was by means of a web browser, using PHP. Using the already mentioned ease of communication between ISCO and PHP, a user could choose an activity instance, check its status (result value, number of executions, etc) and control the progress of the workflow. Figure 3.1 shows the schematics of the application at this point.

At this stage, a prototype was available that allowed for the translation of simple UML activity diagrams to ISCO, their inclusion in the system and posterior instantiation and control through a web-browser and PHP. However, this presents a lack of integration between the modeling process and the application itself and, consequently, the execution of the models.

3.5.2 ArgoUML (and other external editors)

Although initially thought as an option, the coupling of the process in which we generate ISCO code with ArgoUML by means of some sort of plugin

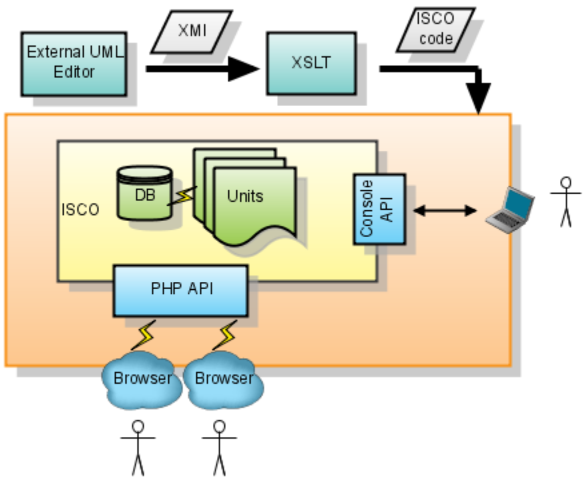


Fig. 3.1: Application Schematics (1)

for ArgoUML, presented an important usability issue: the obligatory use of ArgoUML to perform the modelling.

It is clear that this sort of commitment with ArgoUML is not the right path to follow, as ArgoUML would still be again immediately isolated from this step forward, that is, its collaboration with the application would always be limited to modeling and code generation, and there would not be any kind of communication from the application to ArgoUML, so that any necessary changes to the model would necessarily mean a new model being inserted in the system.

A decision was thus made that the modeling process should not be limited or better adapted to only one external editor, but rather to any that supported XMI output, although still not eliminating the apparent problem of communicating with the editor from the application.

To sum up this section, it should be clear that the motivation to work with an external editor, when it could involve the development of particular modules, was being questioned. The lack of integration was considerable, and any attempt to reduce it towards any application would mean getting further away from any other. It became clear that *integration* was an important factor to ponder.

3.5.3 ISCO and Web2.0

From the previous section comes the thesis that all the modeling and editing should be done in closer contact with the application, allowing, for example, that the need for any adjustment could be solved with the knowledge of what models are present in the system, and that those adjustments could be directly reflected back into the system, if not done directly over a determined model.

The best way to do this is to extend the initial purpose - controlling the engine from a browser - and build an environment where workflow modeling can also happen from within the browser. This approach, although implicating a considerable amount of development, enables us to build a totally integrated web platform where, upon development of several modules, one can model, edit, manage and execute workflow instances, using always the same environment.

However, enabling workflow modeling outside an UML environment raises a relevant question: should we or should we not continue to model using UML DA notation? The adequacy of UML DA to express workflows not being the issue, as we have seen in section 3.1, we believe that the use of UML to model Business Processes benefits from the language's generalized use, as it is a familiar ground.

Web2.0 Technology

The technology we use is AJAX and JSON (Javascript Serialized Object Notation), which together comprise AJAJ. Instead of using isolated libraries or packages we also make use of the Dojo Toolkit [10].

AJAJ allows us to maintain the much needed communication with ISCO through PHP, adding that it can be done asynchronously, meaning, for example, that information can be fetched from or sent to a server without any need of page reloads. The Dojo Toolkit implements a set of graphical capabilities with much interest to the objectives at hand, for example, generation of SVG (Scalable Vector Graphics) shapes on the browser, with drag&drop and matrix manipulation functions.

Figure 3.2 illustrates the schematics of the application in this phase.

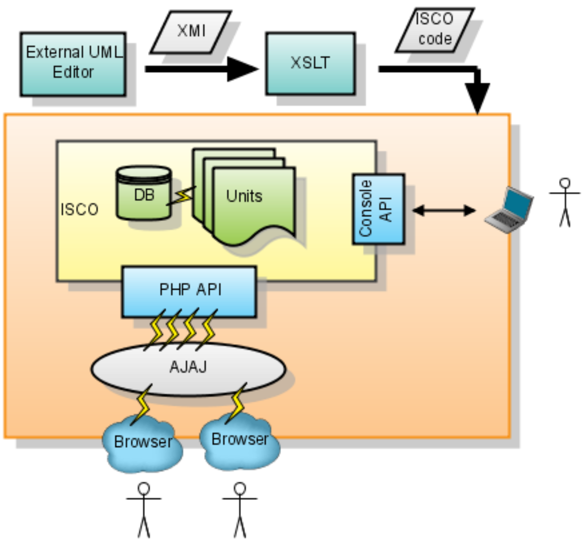


Fig. 3.2: Application Schematics (2)

3.6 External events and web-services

In this section we describe in detail the measures taken to make available the mechanism for calling and returning results from external events, as well as the inner workings of the system when such an event is triggered.

An *external event* is an action that takes place outside our system. All we need to know about these events is how to call them. How or what it will execute is another different matter, irrelevant to our system, as long as a result is effectively returned.

Another important thing about external events is how to catch their results. We to do this by means of a web-service [36]. An easy API is defined that enables simple communication between the system and the (any) external platform, at the expense of having the remote host create the simple necessary web-service capabilities.

The technology in use is SOAP (section 2.6). The client was implemented both in Perl and PHP, to test platform independence, while the server was implemented in PHP alone, due to the need of interaction with ISCO.

3.6.1 Web-services: How actions take place

When an action state is initiated by the flow control, its event is triggered. To "trigger the event" means to access the unit [3] where the actual action of the event is coded. This is true if the event is an internal event, whereas the action is executed and the result is put back in the right place, where it can be accessed by the flow control. If the event is an external event, the mentioned unit will not contain the actual code of the event, but will instead have the needed parameters to issue a call. This call is done from within the core system, by means of a *popen/3*, directed to the wanted call script (Perl, PHP, etc). That call script is our client code[4].

On the remote location, a compatible server will be expecting the call, which will trigger the process, whatever it may be. Despite the fact that an instant answer to this call is actually sent back from the remote server, it will not be the result of the event execution, but merely a signal that the order of execution has been given. The reason why this is is due to uncertainty: we simply do not know what will take place remotely nor how long it will take, and thus cannot "wait still" for an answer to be returned. What we do know is that, at some point, an answer or result of whatever took place will be ready to be sent back. That is why we break this process in two ordered but independent stages.

When ready, a compatible client located on the remote machine will access our web-services server, sending the result of the event. Our server then takes the answer and inserts it into the core system.

[3]Consult section 4.2

[4]The call script will change for each different remote call so, for the time being, we have a client for each remote event.

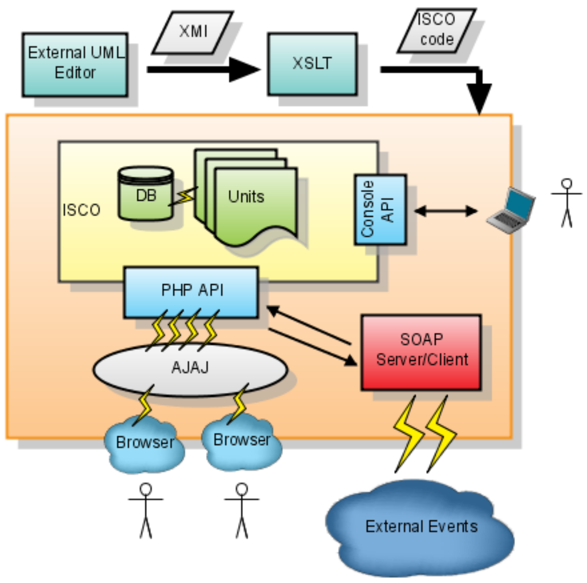


Fig. 3.3: Application Schematics (3).

3.6.2 Web-Service APIs

The remote SOAP server

To make a remote call we use a SOAP client. For this to work, the remote computer must be able to acknowledge and execute our request, by also making available a simple web-service.This web-service should obey the format shown in table 3.1.

The local SOAP server

As mentioned, we make available a SOAP server, for receiving the results of the external events. For accessing the server, the remote host must be able to perform a web-service call with the format show on table 3.2.

Figure 3.4 illustrates the pseudo-code for the server.

	`event_call(f_name, workflow, event)`
`f_name`	*f_name* means *function name*, although this is just an identifier for the action to be executed remotely, be it a function or something else.
`workflow`	This is simply the identifier (*key_id*) of the workflow instance from where the call is originated. The knowledge of this identifier is important to the remote host as it will need to know where to put the final result of the action.
`event`	This is once again an identifier, this time referring the event instance. This reference is important for the same reasons as described on the previous argument.

Table 3.1: Remote Web Service API

```
Load soap library;

Create server;
Configure the WSDL service;
Register the function submit_result;

// Submits the received Result onto ISCO
function submit_result(Instace_id, Event_instance_id, Result)
{...}
```

Fig. 3.4: SOAP server pseudo code.

3.7 Transition Guards

Since we can set up transitions on the fly, we also need to be able to define conditions (or guards) to use with *Conditional Transitions*, directly and immediately when we create these transitions.

For this purpose we've created a simple grammar that allows the user to specify what rule guards the transition. At the moment, this grammar is very simple, but with further refinements it will be extended to allow more complex conditions (see section 7.2.6). Figure 3.5 shows this simple grammar.

As we can see, this is quite a basic grammar which only allows binary comparison between the result of the previous activity (*[RESULT]*) and a value (*value*) of type Integer or String. Also, and for now, the *Result* operand

`submit_result(workflow, event, result)`	
`workflow`	The workflow instance ID.
`event`	The ID of the event instance.
`result`	The result value of the remote event.

Table 3.2: Local SOAP server API

```
s --> [RESULT], binop, value

binop --> [==]
binop --> [=>]
binop --> [<=]
binop --> [>]
binop --> [<]
binop --> [!=]

value --> [INT]
value --> [STRING]
```

Fig. 3.5: Simple grammar for `Condition` building.

must be the first (leftmost) faciend.

Examples of valid transitions are:

Result == 'approved'
Result <= 65
Result != '2007-10-01'

At the system level, the condition gets decomposed and the *Result* operand is replaced by a variable, which will receive a value at execution time.

3.8 Summary

In this chapter we have been through the methods and procedures used for both the decision and accomplishment of the intermediate objectives dealt with in the project. We have discussed the use of UML, an UML editor (ArgoUML), and how its XMI code output posed a suitable bridge from UML to ISCO. We presented the reasons as to why and how we drifted away from the initial goal of the project. The application's ISCO based data structure was shown, as well as its types of persistent information. The API with which the user can start the execution of new workflow instances, or move forward on any existent execution, was covered, as were the stages of the project as the approach shifted from an external UML editor to an integrated editor. We saw how AJAJ and the Dojo Toolkit found their place in the project, and explained *Internal* and *External* events, as well as the role of web-services, through SOAP, in handling the later. Also, we have depicted the simple grammar on top of which *transition guards* are built.

Chapter 4

Data model and representation

In this section we present the adopted data model, as well as several aspects of the internal representation of information.

4.1 Class model

The description of the data model is divided in two parts: one concerning the part of the data model that deals with the modeling information (activities, transitions, positions, etc) and one concerning to the part that deals with the dynamic or execution information. Various ISCO code snippets are also presented. A complete relational model is also available in appendix (page 103.

4.1.1 Modeling data (description)

Next we describe each of the modeling data components and their purpose.

Activities We call *activities* to all nodes, be they control nodes (*join*, *fork*, *start*, *end*, *split* or *decision* and *merge*)or action nodes (*action state* or *simple activity* and *compound activity*). Support for the “flow final” activity is not supported at the moment.

Conditions Theoretically, each “split” activity has an attached condition, responsible for the choice of the path to be followed. For this to happen, the several outcomes of the condition evaluation would have to be kept in a dynamic structure[1]. Since each of the choices are graphically represented with a transition from the split node to the desired target node, we decided to have the decision capability on the transition itself. This way, when the execution flow reaches a split node, we know that all the transitions starting in this node have a condition which needs to be evaluated against the result of the previous action. The evaluation will either be "true" or "false" where, typically, the first "true" to be found indicates the path to follow.

Transitions Used to provide a means for activity transitions, establishing a connection between two activities, transitions store their references. The pair of activities bound to a transition are nicknamed as "source activity" (where the transition begins) and "target activity" (where the transition ends).
There are also two types of transition available: a) a *conditional transition*, which has a guard condition, explained in the previous item, and; b) an *unconditional transition*.

Actions or events Each *action state* (or *simple activity*) has an associated event which is triggered when the execution flow reaches this activity. [2].

Workflows All activities are related to one or more workflows. Also, every execution, or workflow instance, has a parent workflow. Both these cases are supported by the stored workflow information, which will reference all the available workflow models in the system.

Workflow-activity relations As mentioned in the previous item, activities can be used in more than one workflow. Since a workflow is obviously related with several activities, we have a many-to-many relation, which

[1]Being variable in number, the multiple outcomes would have to be stored outside of the *split* itself.

[2]From the possible types of events we are only using the "call event", which is responsible for a function call, and which will always have a "result"

implies that an extra class should be used (meaning an associative database table) to establish this relationship.

4.1.2 Modeling data (ISCO)

Figure 4.1 depicts the hierarchical schema for both activities and transitions, and table 4.2 presents the details of the classes and attributes.

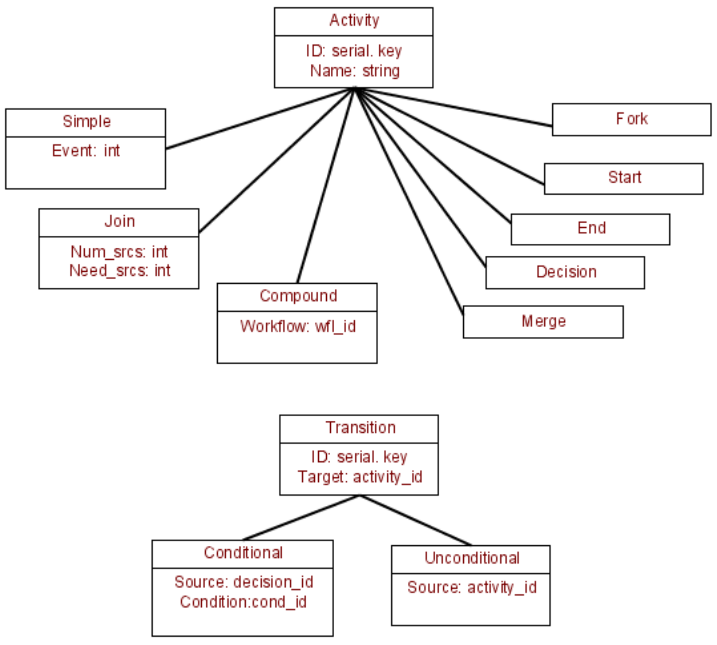


Fig. 4.1: Activities and transitions hierarchical diagram.

ISCO code	Description

`abstract class activity.`
`  key_id: serial. key.`
`  name: text.`
	This is the super-class of all the activities. It contains the following attributes: *key_id*, an automatic sequential key; *name*, the given activity name. This is also an abstract class, as it does not represent any of the activities by itself, which are all specializations of this super class.

Continued on next page

ISCO code	Description

`mutable class activity_simple:`
`                    activity.`
`  event: int.`
	This class represents an “action state”, or “simple activity”. This class is a specialization of *activity*, and inherits both attributes of this class, further adding a new one, *event*, which connects the activity with the event to be performed. Despite being used to reference an object of class event, the *event* attribute is of type *int* - rather than of type *event.key_id* - due to a temporary implementation detail, where it is allowed to create an action state without having an event immediately bound to it.

`mutable class activity_join:`
`                  activity.`
`  needed: int.`
	This class is also a specialization of the *activity* class, and represents the synchronization activity *join*. Besides inheriting both its super-class attributes, the *activity_join* adds one extra attribute, **needed**, which states how many incoming transitions will have to be available for the join to be activated, that is, for it to allow the control flux to proceed.

`mutable class activity_fork:`
`                  activity.`
	This class is another specialization of the *activity* class, and represents the *parallel split* node, also known as *fork*. Just like the previous two, this class inherits both its super class attributes, and adds none other.

Continued on next page

ISCO code	Description
`mutable class activity_decision: activity.`	Also a specialization of the *activity* class, this class represents the exclusive decision *split* node. This class adds no further attributes.
`mutable class activity_merge: activity.`	The *activity_merge* class is another specialization of *activity* and represents the *simple merge* node. No attribute is added.
`mutable class activity_start: activity.`	Also a specialization of *activity*, this class represents the beginning of the control flow. No attribute is added.
`mutable class activity_end: activity.`	Once more, a specialization of *activity*, this class represents the end of the control flow. No attributes are added.
`mutable class activity_compound: activity. act_workflow: workflow.key_id.`	This class represents the compound or non-atomic nodes, and is once more a specialization of *activity*. The **act_workflow** attribute points directly to another workflow, to which the flow control is temporarily assigned upon execution of an instance of this class.

Continued on next page

ISCO code	Description

`abstract class transition.`
`  key_id: serial. key.`
`  target: activity.key_id.`
	The *transition* class is the super class of the two possible transition object types, namely the *transition_conditional* and the *transition_unconditional*. Being abstract, this class makes available two attributes to be inherited by their sub classes. These attributes are: *key_id*, the sequence key identified attribute and; *target*, which points directly to the key identifier of another *activity* as its target node. Despite the fact that every transition has both source and target nodes, only the later is defined here. This is explained in the following rows.

`mutable class`
`        transition_unconditional:`
`                      transition.`
`  source: activity.key_id.`
	As the name indicates, the *transition_unconditional* class represents an unconditional state change, that is, no condition is tested after the source node is finished. The added attribute, *source*, can bind any kind of activity as the source node of an unconditional transition.

Continued on next page

ISCO code	Description

`mutable class`
`        transition_conditional:`
`                    transition.`
`  source: activity_decision.key_id.`
`  cond: condition.key_id.`
	Just like the previous class, *transition_conditional* is also a sub class of the *transition* abstract class and also adds the attribute *source*. Regarding this attribute, the difference is in its type. As we can see, rather than referencing any *activity* object, it only binds exclusive split nodes (*activity_decision*), that is, a conditional transition can only start on a decision node. Since a conditional transition needs a condition to be evaluated, this class adds another attribute, *cond*, which directly references a *condition* object.

`mutable class condition.`
`  key_id: serial. key.`
`  guard: term.`
	A *condition* object represents the atom that must be evaluated in any conditional transition. A *condition* has its own sequential key identifier, *key_id*, and an extra attribute of type "term", *guard* which contains the guard expression of the condition, in the shape of a Prolog term. A guard term is formed has follows:
	
	`(A==20)-A`
	
	The variable A is instantiated in execution time so that the guard can be directly "called", using a Prolog *call/1*.

Continued on next page

ISCO code	Description

`mutable class event.`
`  key_id: serial. key.`
`  name: text.`
	The *event* class represents the action to be performed upon the execution of an *activity_simple* object. An *event* object contains no more than the name of the event. The remaining data will be dynamic, as it will be contained in a dynamic data class.

`  mutable class workflow.`
`  key_id: serial. key.`
`  name: text.`
	A *workflow* object represents a workflow model present in the system. This allows every activity to be linked with one or more workflows. Apart from having its sequential identifier attribute, *key_id*, a workflow only needs to store its *name*.

`mutable class workflow_activity.`
`  workflow: workflow.key_id.`
`  activity: activity.key_id.`
	As mentioned in the previous class, every activity is connected to one or more workflows, or vice-verse. The class responsible for storing this information is the *workflow_activity* class. As it can be seen in the attributes (*workflow* and *activity*), this class binds a *workflow* object to an *activity* object.

Fig. 4.2: Modeling data (ISCO classes)

4.1.3 Execution data (description)

Next we describe each of the execution data components and their purpose.

Workflow instances For each instance of any workflow that is spawned in the system, an *execution* is created. Each *execution* has a reference to its *workflow*, an execution status flag (signaling for a terminated or running instance), a list of activities for execution (useful for parallel executions) and the linear execution trace, which contains the mention to all activities the flow control stepped through.

Activity instances Every time an activity is reached by the flow control, an *activity instance* is created. This instance serves the purpose of storing the references of its originating activity and workflow instance, as well as how many and which other activity instances led to the present instance. An activity instance also has a flag indicating its state of completion, which helps the execution engine.
All this information not only extends the trace capability, with new kinds of information, but also allows for a better way of graphically seeing the current state of any workflow instance. It is worth to mention that, although we have quite a few different activity types, all their instances are kept inside a single table because, up to this point, this approach has been simple and efficient.

Event instances An event is triggered when an action state is started. To trigger an event is to create an event instance containing all the references to its originating activity instance and its execution. Once finished, the event instance will also store the result of the event execution.

Event execution times For each triggered event, be it local or external, both its "start" and "end" timestamps are safely kept. The fact that these pairs are kept apart from the event instance they relate to, even though we can see it is a one-to-one relation, is due to an implementation detail. Since these timestamps are dealt with by database rules, once an event instance is created, the corresponding database row can not be changed since when the rule is fired the row has not yet been written. This way, the rule rather writes the timestamp in a parallel table, along with the reference to the related event instance.

Result input of external results When an external event result enters the system, it is written to this temporary table. This action triggers several events: the end timestamp is written to the relating event instance's execution times table; the result value of the external event is copied to its event instance reference.

4.1.4 Execution data (ISCO)

ISCO code	**Description**

`mutable class execution.`
`  key_id: serial. key.`
`  workflow: workflow.key_id.`
`  complete: int.`
`  act_list: term.`
`  trace: term.`
	Each *execution* object has five attributes, namely: the key, *key_id*, a sequential identifier; a reference to the workflow model, *workflow*; a flag, *complete*, stating if the instance has reached its end or is still in execution; an activity list, *act_list*, containing the activities due for execution on the next engine step and; the trace, *trace*, a compound Prolog term containing the sequence of activities visited so far in the present execution.

`mutable class activity_instance.`
`  key_id: serial. key.`
`  activity_id: activity.key_id.`
`  execution_id: execution.key_id.`
`  valid: int.`
`  came_from: term.`
`  status: text.`
	The *activity_instance* class represents any activity instance of any sub class of *activity*. Each object contains a sequential identifier (*key_id*), a reference to the activity it instantiates (*activity_id*), a reference to the workflow instance (*execution_id*), the number already taken incoming transitions (*info*), the list of activity instances that led to this instance (*came_from*) and the status of completion of the instance.

Continued on next page

ISCO code	Description

`mutable class event_instance.`
`  key_id: serial. key.`
`  act_ins_id: act_inst.key_id.`
`  exec: exec.key_id.`
`  event: event.key_id.`
`  result_value: term.`
	An event instance object (*event_instance*) represents the instance of an event, be it external or internal. Every event instance has attributes regarding its originating activity instance (*act_inst_id*), as well as its execution instance (*exec*) and its event (*event*). The result of the event instance is kept in a Prolog term (*result_value*), which is set by a triggered rule, overriding the initial temporary value. As one can see, at the moment there is no difference between an internal or an external event instance, though further developments, as the internal event programming module, will likely force the integration of separate classes.

`mutable class event_times.`
`  occurrence: int.`
`  start_time: date.`
`  end_time: date.`
	The class *event_times* represents the pair of time flags associated with every event instance. This association is present by the *occurrence* attribute, although there is no direct bound to the event instance's key. This is due to the same reason as described earlier on the previous section. Both the other attributes, *start_time* and *end_time*, are of type *date*, and store the start and end timestamps of the event instance execution.

Continued on next page

ISCO code	Description

`mutable class fifo_event_buffer.`
`  key_id: serial. key.`
`  event: event_oc_stream.key_id.`
`  workflow_inst: exec.key_id.`
`  result: text.`
	This class represents the results of the event instances that are submitted to the system, each of which having the correspondent event instance (*event*) and workflow instance (*workflow_inst*) as attributes. The result of the executed action that was initially set to 0, will shortly be replaced by *result*.

`mutable class component_position.`
`  component: activity.key_id.`
`  x: int.`
`  y: int.`
	This class represents the coordinate pair which signals the position of an activity object. The attributes are *component*, a direct reference to the activity object it relates to, and x and y, the horizontal and vertical coordinates of the component.

Fig. 4.3: Execution Data (ISCO classes)

4.2 Organization in Units

In this section we present the different files of the data model and engine, and how they are organized. We can easily identify the need for several different components in the project, like the workflow engine, the data structure definition, etc, each with specific goals. The fact that by using Prolog we can adhere to the "unit" representation, where each individual module is represented by a different file, called "unit", seems to perfectly fit our purpose. This way, the units used in the project and their description are:

main This unit is the entry point of every interaction with the system. From this unit, all the requests are channeled to the correct unit. The *main* unit also contains and triggers the Prolog goals to create an execution, as well as to step through the different activities of the execution.

process Inside this unit we code the actions that are triggered by the events inside the workflow execution. For "internal" events, all the actions are programmed inside this unit. For an “external” event, the call is simply a call to a script which will in turn make the call to the implicit service.

model(s) Any number of *model* units may be used, as each is intended to represent a workflow model. These units are typically the final result of the XMI processing, and they are ready to be loaded and used by the system.

interfacePHP This unit contains all the predicates that are responsible for any interaction with the GUI. When a request is sent from the graphical web client, it first enters the *main* unit, which redirects the request to the *interfacePHP* unit. According to the request, the relevant action is executed, and the respective answer is returned. This process is also valid to all of the asynchronous requests that are generated from the GUI.

engine This is the responsible unit for executing of the workflow, that is, the intelligence of the execution.

read_write_data This is responsible for handling any request to the database, be it a *write* or *read* request.

graph Though still in a very embryonic stage, this unit will be responsible for rearranging any model components' positions. This will happen upon request and will automatically rearrange the screen coordinates of all the components with the help of Graphviz procedures.

Figure 4.4 depicts what units are reachable from each of the input/output channels.

4.3 Summary

In this chapter we have seen the aspects of the data model adopted for our application, explaining the difference between *model data* and *execution data*, describing each component and it's purpose, and also examining each class

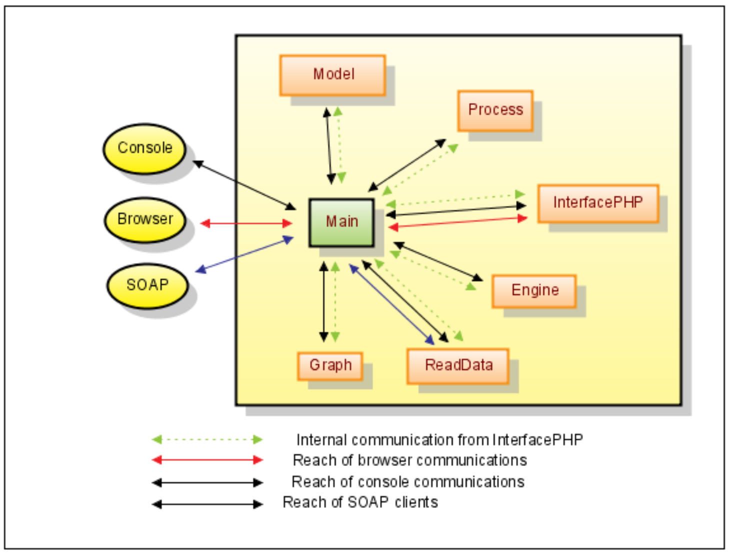

Model
Process
Console
Browser
SOAP
Main
InterfacePHP
Engine
Graph
ReadData
Internal communication from InterfacePHP
Reach of browser communications
Reach of console communications
Reach of SOAP clients

Fig. 4.4: Units schema.

found in the data model. An enumeration of all the *Units* is given, and the organization between them is explained.

Chapter 5

The web client application

In this chapter we describe the application with focus on the graphical component and user interaction, that is, the web browser functionality. In section 5.1 we briefly span through what the web client's intentions are, and how it is organized. Section 5.2 depicts the Workflow Sketcher and the Status Viewer screens of the application, and briefly explains the purpose of each panel and menu. In section 5.3 we go through the several files and organization of the web client, as well as the purpose and function of each of those files. Asynchronous communication with the server is explained in section 5.4 , where both simple and complex requests are covered. Section 5.5 details the web client's API, by explaining the function of each method. Finally, section 5.6 presents two tables where the actual degree of support of AD components and patterns, in the web client, can be seen.

5.1 About the Web Client

The web client is intended to be the main interface between the user and the platform. The client is currently divided in two components: a) the "Workflow Sketcher" page, and b) the "Workflow Status Viewer" page.

The user can define new workflows from scratch, edit a previously created workflow, load instances of any completed workflow, and check the trace and execution status of any ongoing or terminated execution, as well as its activities.

5.2 Graphical structure

Figures 5.1 and 5.2 depict the main workflow sketcher view and the status viewer, respectively.

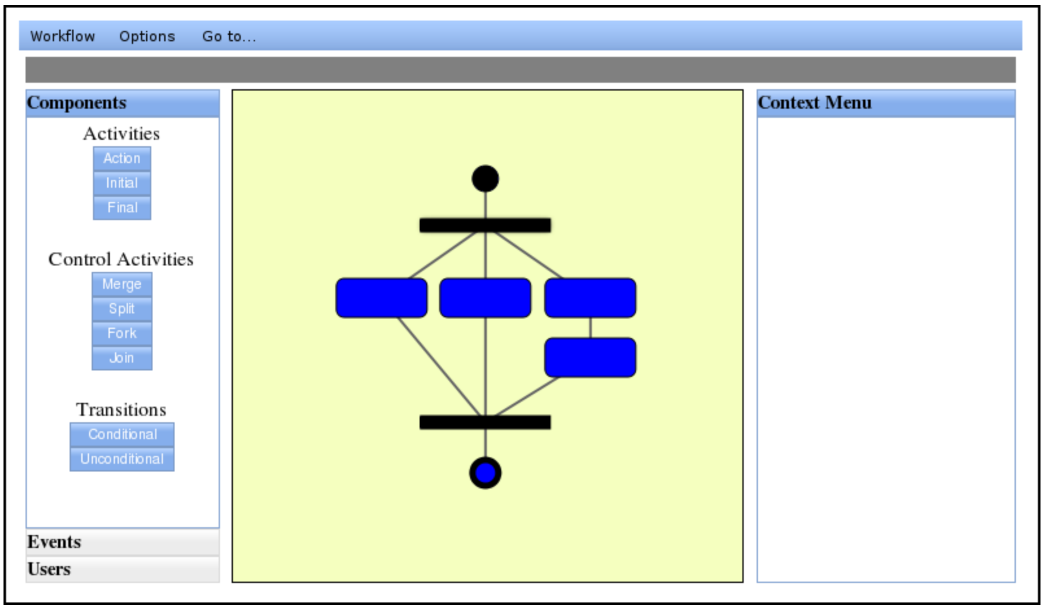


Fig. 5.1: Workflow Sketcher Screenshot

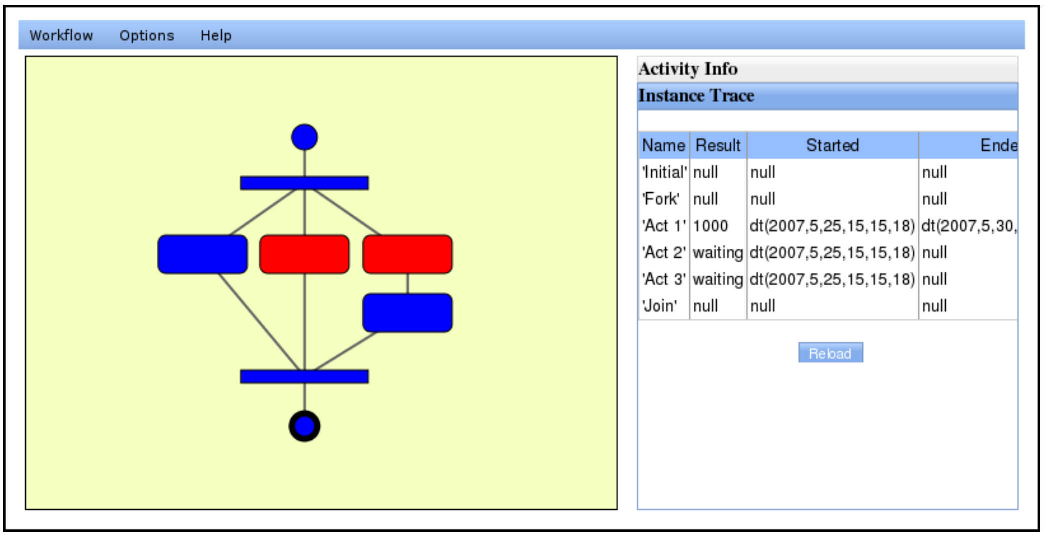
The **Workflow Sketcher** consists of four main sections:

The top menu Where one can create new workflow models, load existent models, configure some options and switch to the Status Viewer.

The left menu The left menu currently comprises three panels: the *components panel* where the user can access the graphical components that constitute the workflow model; the *events panel* where events can be assigned to action activities, and; the *users panel*, which currently allows no actions.

The right menu The right menu (also called the *context menu*) will present information about the selected item on the drawing pane.

The main area Also called the *drawing pane*, this is the area where the graphical components are created, connected and dragged around as a workflow model is built.



Name	Result	Started	Ende
'Initial'	null	null	null
'Fork'	null	null	null
'Act 1'	1000	dt(2007,5,25,15,15,18)	dt(2007,5,30,
'Act 2'	waiting	dt(2007,5,25,15,15,18)	null
'Act 3'	waiting	dt(2007,5,25,15,15,18)	null
'Join'	null	null	null

Fig. 5.2: Status Viewer Screenshot

The **Status Viewer** consists of three areas:

The top menu Allows the user to open a view on an existent workflow execution, as well as configuring some options (currently disabled) and switch to the Workflow Sketcher.

The main pane This is where the loaded workflow execution will appear, with the respective code color for current activated actions. In the status viewer it is not possible to drag&drop the components.

The information panel This panel is formed by two panes: the *activity info* pane shows the execution data of the selected activity on the main pane, and; the *instance trace* which will show the trace of the current execution.

5.3 Files and organization

Our Web Client makes use of a set of functionalities that are interlaced in different files.

First we have `html` files, that serve as normal web pages, and where the information is dynamically display as a reaction to the user's actions and requests. For structural reasons, none of the `html` files have any Javascript scripts or PHP functions, being thus very clean and understandable. We then have `php` files

This section lists the files according to type (`html`, `php`, `js` and `smd`) and explains their purpose and contents.

HTML files As we have only two concepts available in the Web Client - designing workflows and checking their executions' statuses - to each we correspond a different *html* file.

The first, `sketcher.html` allows for all the activities regarding the creation and edition of workflows, that is, positioning shapes, assigning transitions with or without conditions and linking events to activities.

The second `html` file, `status.html`, allows the user to access the trace of the chosen execution, be it running or already concluded.

Both the `html` pages are built based on various Dojo special constructs (called *widgets*), like the drawing pane, menu bars, layout containers, etc. All the dialogs that appear on various occasions are also coded in these files, waiting to be triggered.

JS files These are the files containing the Javascript functions and variables that enable the dynamics of the pages. The files and their respective functions are:

sketcher_funcs.js This file contains all the global variables used in the edition and construction environment, as well as all the functions responsible for: initializing the dynamic Dojo components; drawing draggable shapes (activities) into the drawing pane; connecting the shapes (transitions); creating conditions (transition guards); assign events to the activities. This file is imported by `sketcher.html`.

sketcher_callbacks.js This file contains all the callback functions that are invoked when data is returned from the server to the web client, like when loading stored workflows, loading events or upon the first stage of creating new activities and conditions. Commonly these functions will issue calls to the `sketcher_funcs.js` file. This file is imported by `sketcher.html`.

moves.js This file contains all the handles for the mouse events that are triggered when placing, selecting and moving a shape. This file is imported by `sketcher.html`.

status_funcs.js This file contains all the global variables used in the status checking environment, as well as the functions responsible for: initializing the dynamic Dojo components; drawing shapes and connections on the panel; handling some mouse events. This file is imported by `status.html`.

status_callbacks.js As with `sketcher_callbacks.js`, this file contains all the callback functions that are triggered when data is returned from the server. This file is also imported by `status.html`.

general_funcs.js This file contains the functions responsible for getting information to and from the server, like a list of activities, transitions, events, etc. This file is imported by both `sketcher.html` and `status.html`.

SMD files These files represent JSON remote procedure methods' classes, and allow the developer to easily define the remote procedure call (RPC) type, methods and parameters for remote objects, so that a remote client can easily communicate with that object. Signatures for the procedures are created, which allow for the communication with the functions themselves, in our case, specified in a `PHP` file. We use two distinct `SMD` files: one to take care of the errors, `erroringClass.smd`, and one other where we explicit all the methods used to get or set data in the server, `myClass.smd`.

PHP files As mentioned on the previous item, `PHP` files contain the classes whose method signatures we defined on the `SMD` files. We have two `PHP` files for this purpose: `erroringClass.php` that deals with errors, and; `testClass.php` that defines all the functions that communicate directly with the server.

5.4 Asynchronous requests and callbacks

In this section we explain how the asynchronous contacts with the server are processed and how answers, when expected, are treated.

5.4.1 Simple Requests

Some contacts with the server happen without the need to obtain an answer. These are typically updates on information (like the position of the shapes, names). For example, when an activity is moved on the drawing pane, an asynchronous contact is triggered, to reflect that same change into the server. Figure 5.3 shows a code snippet used to update the coordinates of an activity.

```
...
var myClass = new dojo.rpc.JsonService("myClass.smd");
myClass.update_shape_position(id, x, y);
...
```

Fig. 5.3: Simple Request Code Snippet

Since the file `myClass.smd` correctly defines the `update_shape_position` function with three arguments, the method will be invoked on the respective `PHP` file, which will update the `X` and `Y` coordinates of shape `id` in the server.

5.4.2 Complex Requests

The most usual kind of contact between the web client and the server will require some data to be sent back to the web client, like, for example, the request for a list of all the workflow models available in the system (like depicted in the code snippet in figure 5.4).

```
...
var myClass = new dojo.rpc.JsonService("myClass.smd");
myClass.get_all_workflows().addCallbacks(contentCallBack, contentErrBack);
...
```

Fig. 5.4: Complex Request Code Snippet

As it is shown, the difference between the simple and complex requests is only the addition of the `addCallbacks` function.
From the `PHP` class an array will be returned which will contain the `ID` of the callback, so that it can be dealt with accordingly, and another array containing all the workflows[1].

From here on, the `contentCallBack` function will do what needs to be done depending on the callback `ID`.

5.5 Web Client API

This section exposes each the methods used to communicate from the web client to the server. For each of these methods, its signature is presented, as well as the object returned. One should note that only the structure of the object itself is shown. The full callback object is formed as shown in figure 5.5.

```
array(
        "id" => callback_identification,
        "resultObject" => $RETURN_ARRAY_or_RESULT,
        "message" => message_or_comment );
```

Fig. 5.5: Callback Array Return Object

[1]The objects of this array are, in fact, smaller arrays of two positions, containing the workflow `ID` and `name`.

As it is shown, the structure is formed by a callback identification field, the result field - usually also an associative array, but a single value occasionally - and a message field, which we use for comments and alike.

Following are the descriptions of each method, its signature and return object, organized by its objective (retrieving information, commiting information or creating elements).

5.5.1 *Creation* functions

create_workflow This method requests the creation of an unnamed workflow, whose identifier is then returned. Figure 5.6 shows the signature of this method, while figure 5.7 shows the graphical component that triggers the method.

```
"name":"create_workflow"
```

Fig. 5.6: SMD method signature for creating a new workfow.

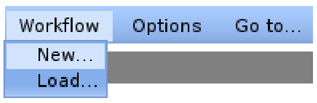


Fig. 5.7: Graphical component used to trigger the creation of a new workflow.

create_activity This method requests the creation of an *empty* activity, that is, an activity with no attributes, of a determined type. Figure 5.8 shows both the method's signature and the return object, while figure 5.9 depicts the menu that triggers the creations.

```
"name":"create_activity",
"parameters":[
{       "name":"act_type", "type":"STRING" } ]
```

```
array(
        "act_type" => "\$type",
        "act_id" => "\$Act_id" );
```

Fig. 5.8: SMD signature (left) and return object (right), for creating an activity.

create_condition This method sends the request for the creation of a new condition, named `name`. The `ID` of the newly created condition is then

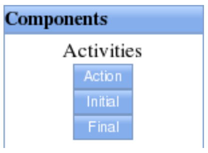


Fig. 5.9: Part of the Component panel, showing some activity creation buttons.

returned to the web client. Figures 5.10 and 5.11 depict respectively the function's signature and graphical dialog box for creating a condition.

```
"name":"create_condition",
"parameters":[
{        "name":"condition",
        "type":"STRING" } ]
```

Fig. 5.10: SMD signature for creating a condition.

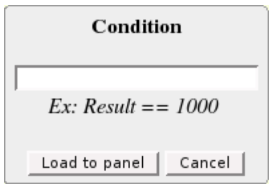


Fig. 5.11: Graphical component for creating a condition.

5.5.2 *Write* functions

set_workflow This method sets the name of a workflow. No object is returned. Figure 5.12 shows the method's signature and figure 5.13 shows the graphical component which triggers this method.

```
"name":"set_workflow",
"parameters":[
{        "name":"workflow_id",
        "type":"STRING" },
{        "name":"name",
        "type":"STRING" } ]
```

Fig. 5.12: SMD signature for setting a workflow's name.

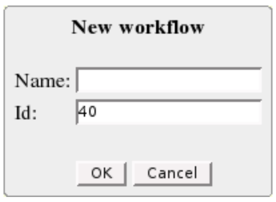


Fig. 5.13: Graphical component showing the definition of a workflow's name.

set_activity This method sends a request which will set up an empty activity with new attributes. Figure 5.14 shows the method's signature and in figure 5.15 the dialog which triggers this method is shown.

```
"name":"set_activity",
"parameters":[
{        "name":"type", "type":"STRING" },
{         "name":"wfl", "type":"STRING" },
{         "name":"name", "type":"STRING" },
{         "name":"id", "type":"STRING" },
{         "name":"x", "type":"STRING" },
{        "name":"y", "type":"STRING" } ]
```

Fig. 5.14: SMD signature for setting an activity's attributes

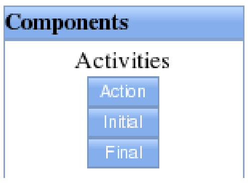


Fig. 5.15: Graphical component for setting a shape's attributes.

update_shape_position This method requests an update on the coordinate attributes of an activity, and is triggered every time an activity is moved. Function signature is shown in figure 5.16.

```
"name":"update_shape_position",
"parameters":[
        {        "name":"id",
                "type":"STRING" },
        {        "name":"x",
                "type":"STRING" },
        {        "name":"y",
                "type":"STRING" }        ]
```

Fig. 5.16: SMD signature for updating a shape position.

assign_event This method assigns an event to a *simple activity*. The method's signature and its graphical use are shown in figures 5.17 and 5.18 respectivelly.

```
"name":"assign_event",
"parameters":[
        {         "name":"activity_id",
                "type":"STRING" },
        {         "name":"event_id",
                "type":"STRING"} ]
```

Fig. 5.17: SMD signature for assigning an event to an activity.

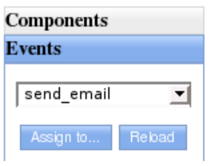


Fig. 5.18: View of the graphical component for event assigning.

assign_transition This method requests the creation of a transition between two activities, graphically connecting them with a line. Only after the callback with the identifier of the created transition is performed will the shapes be connected on-screen. Figure 5.19 shows both the method's signature and return object resulting from its call.

```
"name":"assign_transition",
"parameters":[
{       "name":"shape_1",
        "type":"STRING" },
{       "name":"shape_2",
        "type":"STRING" } ]
```

```
array(
        "t_id" => "\$Id",
        "source" => "\$shape_1",
        "target" => "\$shape_2" );
```

Fig. 5.19: Method API: SMD signature (top) and return object (bottom), from assigning a transition between activities

assign_cond_transition This method performs the same request as the previous, with the difference that the created transition will be a *Conditional Transition*. The corresponding signature and return object can be seen in figure 5.20.

```
"name":"assign_cond_transition",
"parameters":[
{       "name":"condition",
        "type":"STRING" },
{       "name":"shape_1",
        "type":"STRING" },
{ "name":"shape_2",
        "type":"STRING" } ]
```

```
array(
        "t_id" => "\$Id",
        "source" => "\$shape_1",
        "target" => "\$shape_2" );
```

Fig. 5.20: SMD signature (top) and return object (bottom), from assigning a conditional transition between activities

rearrange This method requests an automatic rearrange of the positions of the activities of a workflow. This method is currently disabled.

```
"name":"rearrange",
"parameters":[
{       "name":"workflow_id",
       "type":"STRING" } ]
```

Fig. 5.21: SMD signature for a workflow rearranging call.

5.5.3 *Read* functions

get_all_workflows This method requests a list with all the workflows in the system. Figure 5.22 shows the method's signature and return object, while figure 5.23 shows the graphical component that triggers this function.

```
"name":"get_all_workflows"
```

```
array(
        "wfl_id" => "\$Wfl_id",
        "wfl_name" => "\$Wfl_name" );
```

Fig. 5.22: SMD signature (up) and return object (bottom) for loading the list of workflows in the system.

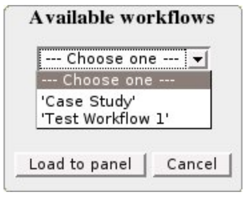


Fig. 5.23: Dialog box showing the loaded workflows.

load_activities This method requests all the activities belonging to a determined workflow. Signature and return object can be seen in figure 5.24 while a screenshot of loaded activities is shown in figure 5.25

```
"name":"load_activities",
"parameters":[
{       "name":"workflow_id",
       "type":"STRING" }       ]
```

```
array(
        "type" => "\$Type",
        "key" => "\$Key",
        "name" => "\$Name",
        "x" => "\$X",
        "y" => "\$Y" );
```

Fig. 5.24: SMD signature (left) and return object (right) for loading all activities of a workflow.

get_open_activities This method requests the list of open activities (activities waiting for their results) of a workflow instance. The result

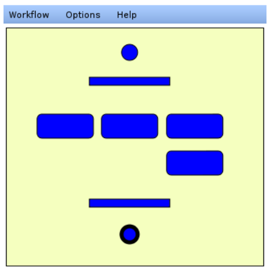


Fig. 5.25: Screenshot after loading the activities of a workflow instance

object of this method is an array of activity identifiers. The method's signature is shown on figure 5.26 and its practical result is shown in figure 5.27.

```
"name":"get_open_activities",
"parameters":[
{         "name":"instance_id",
        "type":"STRING"
        } ]
```

Fig. 5.26: List all open activities of an instance.

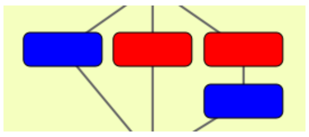


Fig. 5.27: Visualization of an execution with open activities, signaled in red.

get_activity_result This method requests the result value of an activity in a particular execution. In case the activity is yet to terminate, the `null` value is returned. Figure 5.28 presents the signature for this method.

```
"name":"get_activity_result",
"parameters":[
{        "name":"act_id",
        "type":"STRING" },
{        "name":"workflow_id",
        "type":"STRING" },
{        "name":"instance_id",
        "type":"STRING" }        ]
```

Fig. 5.28: SMD signature for returning an activity's result.

get_events This method requests a list with all the events in the system. Figure 5.29 shows the SMD signature and return object of this method and in figure 5.30 shows a graphical component exibiting the loaded events.

```
"name":"get_events"
```

```
array(
        "event_id" => "\$Event_id",
        "event_name" => "\$Event_name",
        "event_term" => "\$Event_term" );
```

Fig. 5.29: SMD signature (top) and return object (bottom) for returning a list of events.

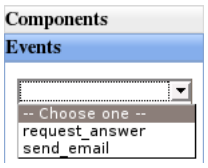


Fig. 5.30: Graphical component showing the loaded events.

load_transitions This method requests all the transitions a workflow. The returned objects contain three identifiers: transition, source activity and target activity. Figure 5.31 shows both the method's signature and return object.

```
"name":"load_transitions",
"parameters":[
{        "name":"workflow_id",
        "type":"STRING" } ]
```

```
array(
        "t_id" => "\$Id",
        "source" => "\$From",
        "target" => "\$To" );
```

Fig. 5.31: SMD signature (top) and return object (bottom) for loading an instance's transitions

reload_trace This method requests a trace reload from a workflow instance. Figure 5.32 shows both the signature and the return object of this method. A table containing a trace can be seen in figure 5.2.

```
"name":"reload_trace",
"parameters":[
{        "name":"workflow_id",
        "type":"STRING" },
{        "name":"instance_id",
        "type":"STRING" } ]
```

```
array(
        "act_id" => "$Act_id",
        "act_name" => "$Act_name",
        "act_result" => "$Act_res",
        "act_start_time" => "$Ev_start_time",
        "act_end_time" => "$Ev_end_time" );
```

Fig. 5.32: SMD signature (top) and return object (bottom) of the reload trace method.

load_executions This method requests a list of all the instances of a determined workflow. Figure 5.33 shows the signature and return object of this method. In figure 5.34 a dialog box is shown, dynamically generated using the information of this method.

```
"name":"load_executions",
"parameters":[
        {        "name":"workflow_id",
                "type":"STRING" } ]
```

```
array(
        "execution_id" => "$Exec",
        "status" => "$Status" );
```

Fig. 5.33: SMD signature (left) and return object (right) resulting of loading a workflow's instances

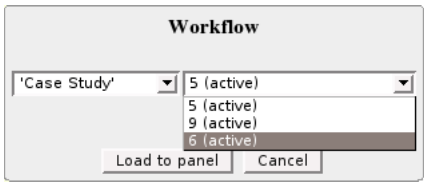


Fig. 5.34: Dialog box showing a list of loaded workflow instances.

5.6 Graphical Support

In this section we will briefly show the state of graphical support for the workflow components and patterns.

5.6.1 Activity Diagram Components

Table 5.1 shows all the graphical components that are present in a UML Activity Diagram and indicates whether or not they are supported in our web-client.

5.6.2 Patterns

As control flow patterns sometimes have graphical requisites to be represented, this section presents the support state of all the patterns regarding our web-client. Table 5.2 shows whether a pattern is or not visually supported.

Component	**Supported**
Initial	Yes
Activity Final	Yes
Flow Final	No
Action	Yes
Join/Fork	Yes
Merge/Split	Yes
Object	No
Send/Receive Signal	No
Flow Arrow	Yes (Though no arrowheads are available at the moment)
Regions	No
Labels	No

Table 5.1: UML AD Component Support

Pattern Name	**Supported**
Sequence	Yes
Parallel Split	Yes
Synchronization	Yes
Exclusive Choice	Yes
Simple Merge	Yes
Multiple Choice	No
Multiple Merge	Yes
Synchronizing Merge	No
Discriminator	Yes
N out of M Join	Yes
Arbitrary Cycles	Yes
Implicit Termination	No
MI with A Priory Design Time Knowledge	No
MI with A Priory Runtime Knowledge	No
MI with no a priory knowledge	No
MI requiring Synchronization	No

Table 5.2: Degree of graphical support for control patterns

Pattern Name	Supported
Deferred Choice	No
Interleaved Parallel Routing	No
Milestone	No
Cancel Activity	No
Cancel Case	No

Table 5.2: Degree of graphical support for control patterns

As we can see in table 5.2, a few AD components are currently left out of the web client's scope which, in turn, makes it impossible for some of the patterns to be implemented.

In the particular case of the *Label* component, although being downright important in UML AD notation as it provides important visual cues as to what each action or transition represents, its use was intentionally put aside. The reason is that, at the system level, the existence of this component is irrelevant, as it does not add any information relevant for the execution. Nonetheless, we do make note that due to its visual importance, from the user point of view, its implementation will not be overlooked in future versions.

As for the remaining components, the choice of leaving them aside of implementation was simply based on the fact that the more common components were enough as a proof of concept. This way, further introductions of these components are programmed for the next iterations of development.

5.7 Summary

In this chapter we have shown the web application component of the project, as well as all of its internal components. We briefly span through what the web client's intention is, and how it is organized. The *Workflow Sketcher* and the *Status Viewer* screens of the application are depicted and explained and we also go through the several files and organization of the web client, their purpose and function. Asynchronous communication with the server is explained, and both simple and complex requests are covered. We present the details of the web client's API, as well as the degree of support, by the web client, of AD components and patterns.

Chapter 6

Case Study

This chapter presents a case study of an (overly simplified) item acquisition process.

In section 6.1 a sketch of the desired UML AD is depicted, and in section 6.2 the workflow is created and its activities and transitions are put together using the *Workflow Sketcher* component. Section 6.3 exemplifies how to code internal and external events and, when created, it explains how they can be assigned to activities, using once again the *Workflow Sketcher* component. In section 6.4 we exemplify the instantiation of the model, and follow by executing some steps (activities) of the workflow. Section 6.5 illustrates how to use the *Status Viewer* component to visualize the execution state. Finally, in section 6.6 we briefly compare some aspects of our application with three other applications.

6.1 Test Workflow

In this section we can see the sketch of the workflow we want to model, which was made using the most typical and simple workflow notation.

The workflow begins with an activity which instantiates a request. The request is then sent to obtain approval of the the department head. If the request is denied, then a fork splits the control flow so that the user is informed and the request gets archived. The control flow is then "joined" and terminated. If the request is approved, a list of suppliers is built and the suppliers are then contacted for a budget request. From between the received budgets, the best offer is then selected or, if no offer is good enough, new suppliers are added to the system, from where the list is built again. When a budget is selected, it will be again subjected to approval. If approved, an order will be placed, if not, the fork section of the workflow is reused, informing the user and archiving the request. Both the options lead to the *final* state.

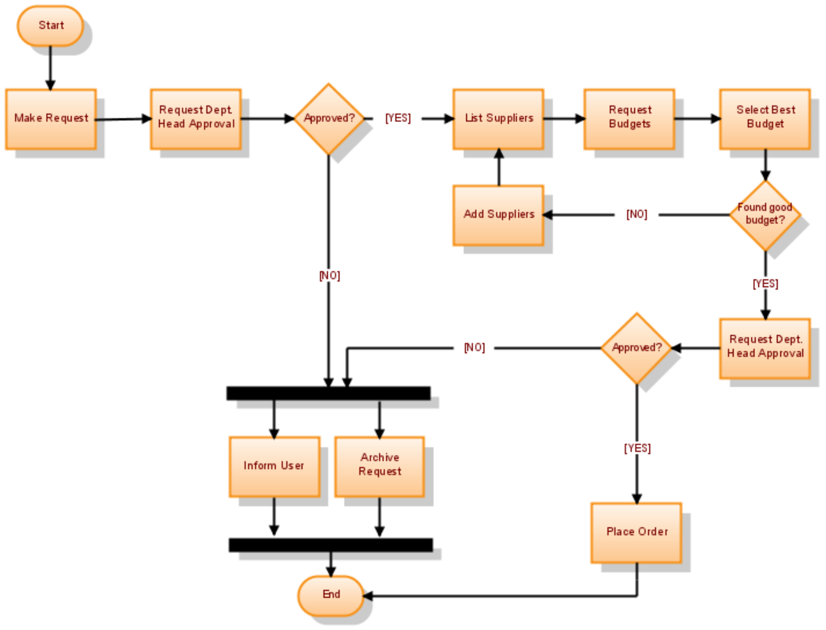


Fig. 6.1: Case study workflow

6.2 Creating workflows, activities and transitions

This sections demonstrates all the steps needed to create new workflows, create new activities and connect them with transitions.

6.2.1 Workflows

The process of creating a new workflow starts by accessing the web client, through a web browser, and selecting the option ***[New...]*** under the menu bar item ***Workflow***.

A dialog appears showing the ID number of the workflow and requesting its name. After entering the workflow name we are ready to begin the modelling.

6.2.2 Activities

Each shape can be added by clicking on the respective button placed on the left side panel. For every activity button pressed a dialog will appear showing a few activity details and requesting the activity name. Note that the activity will only appear on the modeling pane after the name is provided in the dialog box.

By default, all shapes will appear in the upper left corner, and can be dragged around the pane to the desired location.

For our acquisition process we will add all the activities first, following the procedure described earlier, and add the transitions later.

Figure 6.2 shows a portion of the "Workflow Sketcher" with all the activities placed where desired. Although there are letters inside of each action activity, they were manually added to allow better understanding of the model. These letters are related with the activity names in table 6.1.

A - Make Request	**G** - Select Best Budget
B - Ask Dep. Head Approval	**H** - Add New Suppliers
C - Inform User	**I** - Send Request For Head Approval
D - Archive	**J** - Place Order
E - List Suppliers	**F** - Request Supplier's Budget

Table 6.1: Activity-Name Correspondence

As we can see, there are more than one *action states* called *Request Head Dept. Approval.* Although we still have no way of copying&pasting an activity, we forced the scenario so that the reader understands that it can be

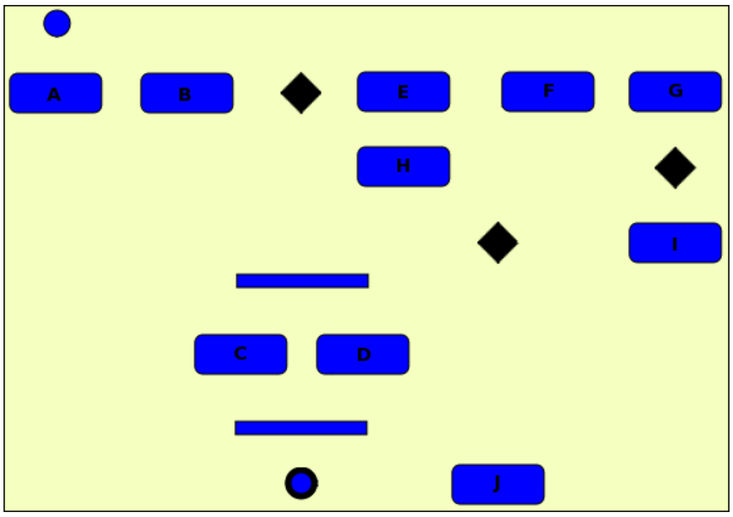


Fig. 6.2: Case study activities

useful to have duplicates of some activities. On this particular case, the duplicate is fake, as it is just another activity with the same name, to which the same event assigned to.

6.2.3 Transitions

With all the activities in place we are now going to add transitions to connect them.

To connect two shapes, we use one of the two transition buttons also available in the left side panel. The choice of which to use depends on which type of transition we want to use (*conditional* of *unconditional*).

We will first connect the *Start* node to the first *Action* node, namely the *A* node (*Make Request*). Clicking on the *Unconditional* button on the side bar, we can see that some of the shapes change their color from the usual blue to green. These are the shapes from where we can have an outgoing *unconditional* transition[1]. All that needs to be done now is to click on both the source and target activities. When clicking on an activity, its border

[1] *Split* and *Merge* nodes do not change their colors, at the moment.

will turn red, indicating that it's selected. When both activities are selected all the colors of the shapes will return to normal, and there will now be a line connecting the two selected shapes. We repeat this process for all the *unconditional* transitions we wish to add.

As for the creation of *conditional* transitions, the process is identical to the described above, but with the difference that the user will now need to specify a condition which will regulate the control flow on the transition. Once we click on the *Conditional* transition button, on the left panel, a dialog box appears for the guard rule to be entered. This kind of transition also has different types of starting activities, which will "signal" different shapes as possible starting points for the transition.

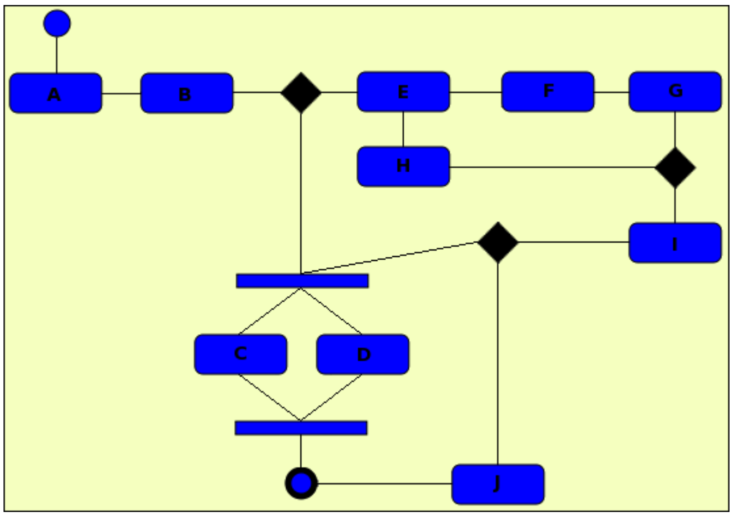


Fig. 6.3: Case study activities and transitions

For now, there are no direction indicators on the transitions. Between activities `E`, `F`, `G` and `H` there is a cycle, which is visually ambiguous without the directed transitions. The reader should interpret the control flow regarding the alphabetic order of the labels, that is, from activity `E` there is only one transition (to activity `F`) and from activity `H` the control flow goes to activity `E`.

At this moment, all the changes made in the web-client have been communicated to the server and made persistent, so no saving is needed.

6.3 Event programming

This section describes how *Internal* and *External Events* can be programmed[2].

It is worth to note that, while some events are clearly to be implemented as internal events and others are to be left to be external events, this does depend on the planned execution architecture and which services are available.

For this case study we assume we have contact with the suppliers, that is, that we can use our web-service capabilities, which will allow us to send requests and wait for the answer.

It is also worth to note that, since we do not have either user or role management capacity, just as no graphical execution module, interaction with the user is very limited, and not role or user identity based. This way, these interactions only occur inside the text console where we are "executing" the workflow instance.

6.3.1 Programming Internal Events

An internal event can be, for example, a text form validation, an arithmetic operation, the sending of an e-mail, etc. In our use case, almost all the activities will trigger internal events except for two of them: "Request Supplier Budget" and "Place Order".

We will now code the internal event of activity "Make Request". To make a request is to create a record with the user data and the request text. Figure 6.4 shows the code of the "Make Request" event.

6.3.2 Programming External Events

As described in section 3.6, the actual event of an external event is not programmed locally, that is, we do not have the responsibility of the execution of the event as it is located somewhere unknown. All we do, from the server

[2]For the time being, programming events requires editing one of the units of the program. This implies that the binary be re-compiled, after all editing is complete.

```
process(make_request) :-
  print('Enter the following data:'),nl,
  print('Name: '), read(Name), nl,
  print('E-mail: '), read(Email), nl,
  print('Request: '), read(Request),nl,nl,
  fifo_event_buffer(Event_occurrence_id, Exec_id,
                   request(Name, Email, Request)) :+ ,
  print('Request processed.'), nl.
```

Fig. 6.4: Internal Event Example

side, is to trigger the local script that will in turn trigger the remote event through a SOAP call.

Figure 6.5 depicts the "Request Supplier Budget" event, which will contact a supplier with an item, or list of items, for which we want a budget.

```
process(request_suplier_budget, Stream) :-
        popen('perl perl_request_sbudget.pl ARGS', read, Stream).
```

Fig. 6.5: External Event Example (pseudocode)

As we can see, we only *popen/3* the SOAP client responsible for this request and it will send the data over to designated supplier.

6.3.3 Assigning events

Now that the events have been created, we can assign them to the correspondent activities. Since there is no difference in assigning external or internal events, we will exemplify by assigning the *Make Request* event to the activity with the same name.

To assign an event, we start by clicking on the *Events* pane, on the left menu. All the events should be automatically loaded and available from the dropdown box, where we will select the *Make Request* event and then click ***[Assign to...]*** button. Now we just need to click on the *Make Request* activity (the activity labelled *A* in figure 6.2) and all is set.

6.4 Instantiation

For the moment, instantiation of workflows, that is, creating new instances for execution, needs to be done from a text console running on same server. This section describes this process.

We start by running the binary file in a text console, with the command shown in figure 6.6.

```
  ~$ ./wfl_bin
GNU Prolog/CX 1.2.18
By Daniel Diaz
Copyright (C) 1999-2006 Daniel Diaz
| ?-
```

Fig. 6.6: Running the binary

One will notice that we are now inside the GNU Prolog top level shell. To create an instance of a workflow, we need to have that workflow's identifier, which we will get by consulting the available workflows in the system (exemplified in figure 6.7).

```
| ?- workflow(Id, Name).

Id = 1
Name = 'Test Workflow 1'

Id = 2
Name = 'Case Study'
```

Fig. 6.7: Workflow list

We can see that there are two workflows in the system. We want to instantiate the workflow *'Case Study'* we'll use the simple predicate described in 3.4, as shown in figure 6.8, which also depicts the output of the instantiation.
From the output we see that an instance of the start activity was created, and consequently executed, as a result of the instantiation. This indicates that the current workflow instance has been started and is ready for execution, which is exactly the next step.

To continue with the execution we use another simple predicate, as shown in figure 6.9.

```
| ?- start(2, Instance_id).

Workflow ID correct: 'Case Study'
Workflow instance created (ID: 1)
Start activity instance created and executed (ID: 42)

Instance_id = 1
```

Fig. 6.8: Workflow Instantiation

```
| ?- step(2, 1).

Workflow ID correct: 'Case Study'.
Found 0 waiting activities.
Found 1 completed activity instance:
 - [instance 42] found 1 transition(s) [to activity ID=4];
```

Fig. 6.9: Workflow Execution

6.5 Checking execution states

Having instantiated and executed a couple of activities on our workflow, we now want to check how it is going. For this, we will access the "Workflow Status Viewer", the other component of our web client.

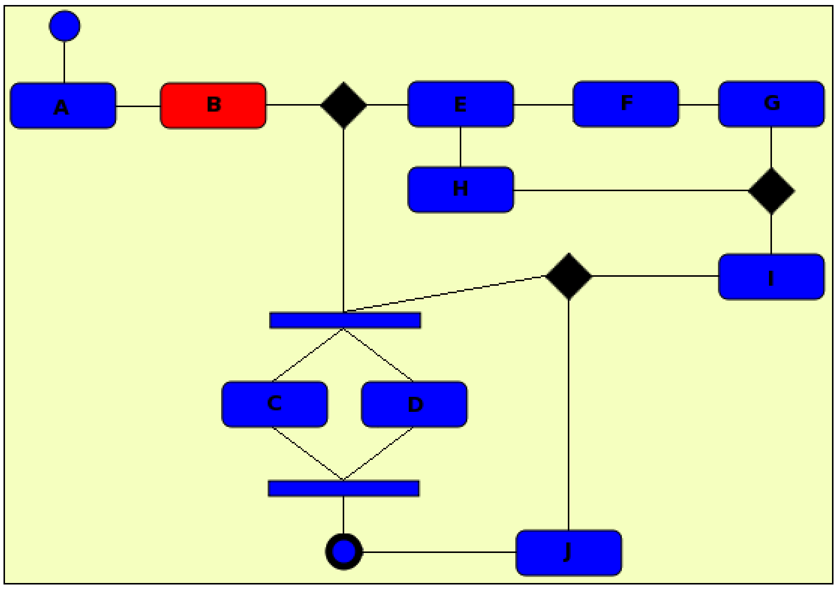


Fig. 6.10: Case study activities and transitions

6.6 Comparison with other software

Figure 6.2 presents a comparison between some aspects of our application and other three, namely, *OpenWFE*, *OSWorkflow* and *OpenFlow*.

	Standalone Workflow Edition GUI	Online Graphical Workflow Edition Client	User Management	Role Management	Event Programming Language	Graphical Execution Status Visualization	Graphical or Online Execution Support	Implementation Language	Execution engine
Our App	No	Yes	No	No	GProlog-cx, Isco	Yes	No	GProlog-cx, Isco	Yes
OpenWFE	No	Yes (Droflo)	Yes	Yes	Python,Perl,Ruby...	Yes	Yes	Java, Ruby	Yes
OSWorkflow	Yes	No	Yes	Yes	Any	No	Yes	JSP	YES
OpenFlow	No	No	Yes	Yes	Python	Yes	Yes	Python	Yes

Table 6.2: Software properties' comparison table

6.7 Summary

On this chapter we have presented a use case for our application, where an acquisition process is modeled. The sketch of the workflow is presented and then modeled in phases, using one of the components of our web-client. We start by defining all the activities that comprise the workflow and then create all the needed transitions. We then exemplify how to program both internal and external events, and then how to assign them to the activities. We also instantiate the workflow and execute a few steps using the console, and then check the execution status using the other component of the web-client. We also present a comparison of some aspects between our application and 3 other applications.

Chapter 7

Conclusions and future work

In this section we review the project's main objectives and conclude upon our work so far, as to the accomplishment of the initial objectives.
Also in this section, we present what we consider to be the relevant future work that could be done having our project as its basis.

7.1 Assessment

The first main objective of this project was to translate UML Activity Diagrams into ISCO code. To do so, we would model the diagrams on a UML editor, export it do XMI and then parse this notation to extract the necessary information to have an ISCO representation.

Although we did manage to digest the XMI representation of the Activity Diagrams and produce valid ISCO code, we realized that the dependency on an external editor raised some usability and integration issues. Regarding usability, every time a correction needed to applied to a workflow, the whole process of exporting, translating and inserting in the workflow engine had to be repeated, assuming the user had saved the UML diagram in the editor in the first place. Concerning integration, suffice it to say that we had two isolated platforms.

Of the set of considered solutions for this problem, we chose to implement our own web-based editor and we are still convinced that it was the wise decision to make, as having an integrated editor allows a better and closer action&reaction from the server.

In regard of the first objective we can say that we have not only accomplished the translation of UML Activity Diagrams into the ISCO language, but also evolved the scope of the project to a wider and more pertinent angle. Nevertheless, we also find it important to mention that the early work with the external UML editor was not put to waste, as it pointed us in the right direction in terms of what to expect from a UML editor, regarding, of course, Activity Diagrams. Further, we do maintain, as an option, the XMI translation tools.

The second main objective of this work was to build a workflow execution system that would downright execute the previously mentioned translations. Taking advantage of the UML Activity Diagram's translation to ISCO, we chose to build this "engine" entirely in the ISCO language. This allows us to work directly with the workflow data and to express complex relations with that data. Although the number of supported workflow patterns is still small, we think that the current prototype sets a firm basis upon which the other workflow patterns can be implemented.

As far as the event module goes, we can conclude that, being the engine a generic workflow execution system, the correct path to follow is to treat every event as an *External Event*. This way, total independence is attained from process or business particular issues.

Still on the subject of events, we are aware that the current process of adding events to the system offers no commodity, and can in fact be troublesome. A better way introduce *Internal Events* and to link *External Events* is thus needed.

The third main objective of this project was to build a tool that would allow the user to monitor the execution of the many instances running in the system at a given time.

We can conclude that the early tests and developments, which involved a dynamic PHP web-page with information from the system, were perfectly suited for the necessities at hand at the time.

But with the mentioned change of scope in the project, those necessities rapidly became obsolete. The choice of having a web-based editor dramatically changed what was expected of the “little” web-client thought at first. Aside from the creation and edition of workflows (as activity diagrams) the current prototype tool also allows that initial monitoring capability, though now benefiting from a different technology, which greatly improves the usability and responsiveness of the interface.

To summarize:

1. we have successfully established the necessary semantics to support activity diagrams as workflows in an ISCO back-end;

2. we have prototyped a mechanism that translates XMI representations of UML Activity Diagrams into the ISCO language;

3. we have successfully implemented a prototype workflow executing system, also in the ISCO language, which executes simple workflows;

4. we have implemented a mechanism that allows an action to be triggered and executed remotely, using the web-service technology;

5. we have implemented a prototype web-client that allows the user to create and edit workflows (using a UML Activity Diagram representation) and also to monitor the state of a determined workflow instance, all done in an asynchronous fashion.

In general, we have put together the basis for an integrated platform for workflow creation, management and execution without leaving the web-browser.

7.2 Future Work

7.2.1 Model Shortcuts

Although we have maintained ourselves on the path of near-strict UML, we consider that the use of "model shortcuts" (like implicit forks and joins) might be appealing, but we also know that the necessary changes at the engine level are quite broad.

So, rather than having these changes implemented at the model level, with downright consequences in the execution and, therefore, the engine, we might consider to have it as a mask. This means that the user will be able to draw with "shortcuts", but they will be interpreted as strict UML by an analysis ran a posteriori.

7.2.2 Patterns

Besides the early stage of development of the major components of this project, we believe that we have proven the point that UML Activity Diagrams can be treated as capable of representing complex workflows and most of their patterns. We further believe that the analysis of these Activity Diagrams can produce capable "system code", that is, they can lead to the specification of all the information needed to have an engine running and executing instances of the models.

Synchronizing merge proposal

As discussed in section 2.2.2, UML ADs offer no solution for the *synchronizing merge* pattern. Using little more than the usual UML AD representation we believe a solution can be achieved.

By allowing the *Merge* node to have a special condition, which can be affected/modified on execution time, we can pass it the information of how many transitions (outgoing from the *Fork*) were activated. At this point, the *Merge* node is capable of waiting only by the desired number of incoming nodes, even if not exactly which. This implies a few changes in the system, namely, allowing the creation of condition instances, as this proposal causes the conditions to be workflow instance dependent, rather than workflow model dependent.

Additionally, and to complete the total capability of the *Synchronizing Merge* pattern, a graph analysis [1] (or similar process) might be performed that will inform the engine if a determined control flow might or might not reach a certain activity. This would almost directly enable the pattern not to wait for a particular control flow that would never reach the merging node.

7.2.3 Execution Component

Although, for the moment, workflow instances can only be executed from a text console, a module is being considered that will allow the execution to be performed in a web environment similar to the modeling's and the viewing's. This environment will include a user and role based system, allowing, for example, that certain activities be directed to a specific user or role group. This module will further approximate this platform to a WfMS (Workflow Management System).

7.2.4 Authentication

Resulting of the specification of the previous section, an authentication module is also in the making which will set the permissions throughout not only the execution system, but all the platform, thus enforcing an overall role based policy for all the users of the system.

7.2.5 Logical Model Benefits

By using ISCO, logical data analysis becomes inexpensive. Based on this premiss, modules are being considered that can perform various types of analysis over the data stored [2].

One of these modules is the *Execution Statistics Module*, which will enable a view over a determined workflow, where statistical information is displayed

[1]Operations concerning graphs tend to be particularly straightforward to implement in Prolog.

[2]The validity of these modules is still to be decided.

so that design pitfalls can be detected, like places where execution hangs most of the times, control flow paths that are never taken, etc.

Another one of these modules is the *Workflow Analogy Module*, which will allow for workflow similarities to be detected, thus possibly encountering equivalent/identical workflows in the system (whether graphically or functionally).

7.2.6 Transition guard grammar extension

As seen in 3.7, the current grammar allowed for defining transition guards is very limited. The extension of this grammar, combined with other modules would represent a better way to define more capable and complex processes. Combined, for example, with the capability to address to a pool of event results, would enable a choice to be made further down the control flow based on any of the results previously obtained on the execution.

Appendix

Database Structure Description

Figure 1 depicts the entity-relation model of the database, followed by the database structure and respective details.

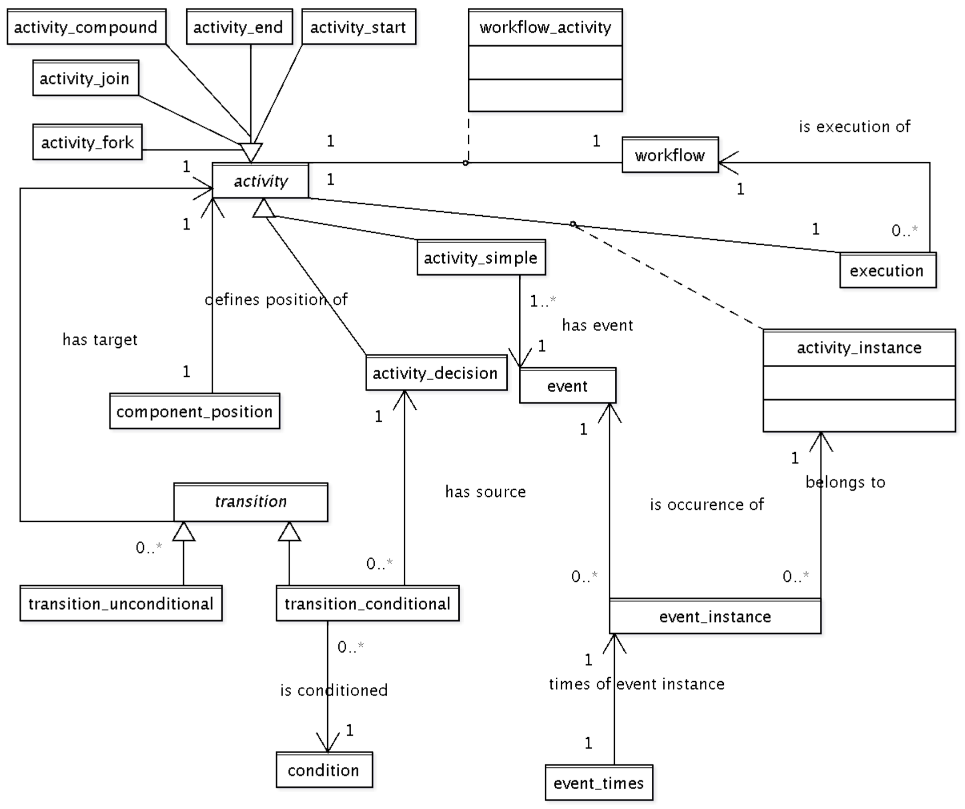


Fig. 1: Entity-Relation Model

Table: activity

activity Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__activity_key_id'::text)*
	name	text	

Table: activity_compound

activity_compound Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__activity_key_id'::text)*
	name	text	
workflow.key_id	act_workflow	integer	

Table: activity_decision

activity_decision Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__activity_key_id'::text)*
	name	text	

Tables referencing this one via Foreign Key Constraints:

- transition_conditional

Table: activity_end

activity_end Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__activity_key_id'::text)*
	name	text	

Table: activity_fork

activity_fork Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__activity_key_id'::text)*
	name	text	

Table: activity_instance

activity_instance Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__activity_instance_key_id'::text)*
	activity_id	integer	
execution.key_id	execution_id	integer	
	info	text	
	came_from	text	
	status	text	

activity_instance Constraints

Name	Constraint
activity_instance_activity_id	CHECK (ok_activity_key_id(activity_id))

Tables referencing this one via Foreign Key Constraints:

- event_occurence_stream
- waiting_join

Table: activity_join

activity_join Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__activity_key_id'::text)*
	name	text	
	needed	integer	

	sources_number	integer	

Table: activity_merge

activity_merge Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__activity_key_id'::text)*
	name	text	

Table: activity_simple

activity_simple Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__activity_key_id'::text)*
	name	text	
	event	integer	

Table: activity_start

activity_start Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__activity_key_id'::text)*
	name	text	

Table: component_position

component_position Structure

F-Key	Name	Type	Description
	component	integer	
	x	integer	
	y	integer	

component_position Constraints

Name	Constraint
component_position_component	CHECK (ok_activity_key_id(component))

Table: condition

condition Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__condition_key_id'::text)*
	guard	text	
	variable	text	

Tables referencing this one via Foreign Key Constraints:

- transition_conditional

Table: event

event Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__event_key_id'::text)*
	name	text	

Table: event_instance

event_instance Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__event_occurence_stream_key_id'::text)*
activity_instance.key_id	activity_instance_id	integer	
execution.key_id	execution	integer	
	event	integer	
	start_at	timestamp without time zone	

	end_at	timestamp without time zone	
	result_value	text	

event_instance Constraints

Name	Constraint
event_instance_event	CHECK (ok_event_key_id(event))

Tables referencing this one via Foreign Key Constraints:

- fifo_event_buffer

Table: event_times

event_times Structure

F-Key	Name	Type	Description
	occurence	integer	
	start_time	timestamp without time zone	
	end_time	timestamp without time zone	

Table: execution

execution Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__execution_key_id'::text)*
workflow.key_id	workflow	integer	
	complete	integer	
	act_list	text	
	state	text	

Tables referencing this one via Foreign Key Constraints:

- activity_instance
- event_occurence_stream
- fifo_event_buffer

Table: fifo_event_buffer

fifo_event_buffer Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__fifo_event_buffer_key_id'::text)*
event_occurence_stream.key_id	event	integer	
execution.key_id	workflow_instance	integer	
	result	text	

Table: transition

transition Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__transition_key_id'::text)*
	target	integer	

transition Constraints

Name	Constraint
transition_target	CHECK (ok_activity_key_id(target))

Table: transition_conditional

transition_conditional Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__transition_key_id'::text)*
	target	integer	
activity_decision.key_id	source	integer	
condition.key_id	cond	integer	

transition_conditional Constraints

Name	Constraint
transition_conditional_target	CHECK (ok_activity_key_id(target))

transition_target	CHECK (ok_activity_key_id(target))

Table: transition_unconditional

transition_unconditional Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__transition_key_id'::text)*
	target	integer	
	source	integer	

transition_unconditional Constraints

Name	Constraint
transition_target	CHECK (ok_activity_key_id(target))
transition_unconditional_source	CHECK (ok_activity_key_id(source))
transition_unconditional_target	CHECK (ok_activity_key_id(target))

Table: workflow

workflow Structure

F-Key	Name	Type	Description
	key_id	integer	*PRIMARY KEY DEFAULT nextval('is__workflow_key_id'::text)*
	name	text	

Tables referencing this one via Foreign Key Constraints:

- activity_compound
- execution
- group_workflow
- user_workflow
- workflow_activity

Table: workflow_activity

workflow_activity Structure

F-Key	Name	Type	Description
workflow.key_id	workflow	integer	
	activity	integer	

workflow_activity Constraints

Name	Constraint
workflow_activity_activity	CHECK (ok_activity_key_id(activity))

Function: ni_activity_key_id(integer)

Returns: bigint

Language: SQL

```
select count(*) from "activity" where "key_id" = $1
```

Function: ni_event_key_id(integer)

Returns: bigint

Language: SQL

```
select count(*) from "event" where "key_id" = $1
```

Function: ni_transition_key_id(integer)

Returns: bigint

Language: SQL

```
select count(*) from "transition" where "key_id" = $1
```

Function: ok_activity_key_id(integer)

Returns: boolean

Language: SQL

```
select ni_activity_key_id($1) = 1
```

Function: ok_event_key_id(integer)

Returns: boolean

Language: SQL

```
select ni_event_key_id($1) = 1
```

Function: ok_transition_key_id(integer)

Returns: boolean

Language: SQL

```
select ni_transition_key_id($1) = 1
```

Bibliography

[1] OMG, "Unified modeling language specification." http://www.omg.org/, 2006. version 2.1.

[2] M. Dumas and A. H. M. ter Hofstede, "Uml activity diagrams as a workflow specification language," in *UML '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, (London, UK), pp. 76–90, Springer-Verlag, 2001.

[3] OMG, "Xml metadata interchange (xmi)." http://www.omg.org/, 2005. Mapping Specification, version 2.1.

[4] M. Boggs and W. Boggs, *Mastering UML with Rational Rose 2002*. Alameda, CA, USA: SYBEX Inc., 2002.

[5] S. Abreu, "Isco: A Practical Language for Heterogeneous Information System Construction," in *Proceedings of INAP'01*, 2001.

[6] R. Gamito, "Modelação de workflows com uml e ferramentas declarativas," in *Proceeding of XATA2007, XML: Aplicações e Tecnologias Associadas (FCUL, Lisboa, 15 e 16 de Fevereiro de 2007)*, February 2007.

[7] C.-H. Tsai, H.-J. Luo, and F.-J. Wang, "Constructing a bpm environment with bpmn," in *FTDCS '07: Proceedings of the 11th IEEE International Workshop on Future Trends of Distributed Computing Systems*, (Washington, DC, USA), pp. 164–172, IEEE Computer Society, 2007.

[8] Y. Foundation, "Yawl - yeat another workflow language." http://yawlfoundation.org. Project Homepage.

[9] J. J. Garrett, "Ajax: A new approach to web applications," February 2005. Seminal.

[10] "DOJO: the javascript toolkit." http://dojotoolkit.org/.

[11] "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)." W3C Recommendation 27 April, 2007, http://www.w3.org/TR/soap12-part1/.

[12] "Simple Object Access Protocol (SOAP) 1.1." W3C Note 08 May, 2000, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[13] "Wikipedia - workflow page." http://en.wikipedia.org/wiki/workflow.

[14] OMG, "Unified modeling language: Superstructure," February 2007. version 2.1.1, (non-change bar).

[15] S. White, "Process modeling notations and workflow patterns," *L. Fisher*, vol. ed. 'Workflow Handbook 2004', Future Strategies Inc., Lighthouse Point, FL, USA., pp. 265–294.

[16] IBM, *IBM MQSeries Workflow Programming Guide: Version 3.3.* Armonk, USA: IBM Corporation, 2001.

[17] Verve, "Verve component workflow engine concepts." Verve, Inc.: San Francisco, CA, USA, 2000.

[18] P. Wohed, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and N. Russell, "Pattern-based analysis of the control-flow perspective of uml activity diagrams.," in *ER* (L. M. L. Delcambre, C. Kop, H. C. Mayr, J. Mylopoulos, and O. Pastor, eds.), vol. 3716 of *Lecture Notes in Computer Science*, pp. 63–78, Springer, 2005.

[19] S. Abreu and V. Nogueira, "Using a Logic Programming Language with Persistence and Contexts," in *Proceedings of the 16th International*

Conference on Applications of Declarative Programming and Knowledge Management (INAP 2005) (M. Umeda and A. Wolf, eds.), (Fukuoka, Japan), Waseda University, October 2005.

[20] L. Monteiro and A. Porto, "Contextual logic programming," in *Logic Programming: Proc. of the Sixth International Conference* (G. Levi and M. Martelli, eds.), pp. 284–299, Cambridge, MA: MIT Press, 1989.

[21] "Apache http server homepage." http://httpd.apache.org/.

[22] "Php homepage." http://www.php.net/.

[23] "Wikipedia - ajax." http://en.wikipedia.org/wiki/Ajax_(programming).

[24] "JSON Project Homepage." http://json.org/.

[25] "JSON: The Fat-Free Alternative to XML." http://www.json.org/xml.html/.

[26] "The AJAX response: XML, HTML, or JSON?." http://www.quirksmode.org/.

[27] "JSON Message." http://ajaxpatterns.org/wiki/index.php?title=JSON Message.

[28] S. Hada and H. Maruyama, "SOAP Security Extensions." http://www.trl.ibm.com/projects/xml/soap/wp/wp.html, November 2000.

[29] "Graphviz - Graph Visualization Software." http://www.graphviz.org/.

[30] "The DOT Language." http://www.graphviz.org/doc/info/lang.html.

[31] A. D. Lucia, R. Francese, and G. Tortora, "Deriving workflow enactment rules from uml activity diagrams: a case study," in *HCC '03: Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, (Washington, DC, USA), pp. 211–218, IEEE Computer Society, 2003.

[32] Tigris.org, “Argouml project home.” http://argouml.tigris.org. Project Homepage.

[33] P. Prescod, “Xslt and scripting languages.” http://www.idealliance.org/papers-xml2001/papers/05-03-06.html.

[34] “Xsl transformations.” http://www.w3.org/TR/2007/REC-xslt20-20070123/. Version 2.0.

[35] E. R. Harold, *XML Bible*, p. 486. New York, NY, USA: John Wiley & Sons, Inc., 2001.

[36] “Web Services Activity Statement.” W3C, http://www.w3.org/2002/ws/Activity.html.