Policy-driven Reconfiguration of Service-targeted Business Processes

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by

Stephen Mark Gorton

Department of Computer Science
University of Leicester

April 2011

Abstract

Workflows are a key part of Business Process Management, offering the potential to automate a number of business activities. Workflows are though constrained to their design, i.e. workflow functionality does not extend outside its own specification. A relatively small number of solutions to this inflexibility have been proposed. However, all approaches so far are either at the orchestration level or are tightly-coupled with the workflow, whereas we consider that the problem is at the business level and needs to be loosely coupled from the workflow.

Significant value can be gained from separating core functionality in a workflow from variability to that core process. Both can be defined separately and yet still corporately execute to provide a variety of execution states that match the given context. Functionality of the workflow can be supplied by Service Oriented Architecture.

Thus we define StPowla as a combination of workflows, policies and Service Oriented Architecture. Workflows define the core business process, policies define the possible variability of the processes and Service Oriented Architecture provides the underlying functionality. We specifically present a set of reconfiguration functions that can be called by policies on workflows and define each of these as graph transformation rules.

We provide an encoding from StPowla processes to SRML models, including core workflow descriptions and variability, in order to make precise the relationship between the constituent parts of StPowla. We apply the StPowla approach to an industrial case study, provided by an industrial partner.

Disclaimer

This work was carried out under the supervision of Dr Stephan Reiff-Marganiec at the University of Leicester, funded by the IST-FET IST-2005-16004 project Sensoria (Software Engineering for Service-Oriented Overlay Computers). All of the work herein has been performed by the author except where otherwise indicated.

Work described in some sections has been previously published, in particular:

- The workflow notation in Chapter 3 has been published in [38] and, more extensively in [39].
- The core policy language customization in Chapter 3 has been published in [36]. This was joint work with Dr Carlo Montangero, Dr Stephan Reiff-Marganiec and Dr Laura Semini. The author's role was to provide the descriptions of the workflow notation and operators. Some initial work was presented in [40] and [37], both principally written by the author.
- The encoding to SRML was published initially in [12] and, more extensively in [13]. The author's role was to provide the policy descriptions and case study scenario. The SRML transitions were principally developed by Dr Laura Bocchi. These were updated by the author in Chapter 6 to reflect the updated case study.

The StPowla aspects of Chapter 6 are solely the author's work. Material presented in Chapters 2, 4 and 7 are solely the author's work.

Stephen Gorton

Leicester, April 2011

O God and Heavenly Father,

Grant to us the serenity of mind to accept that which cannot be changed;

courage to change that which can be changed,

and wisdom to know the one from the other,

through Jesus Christ our Lord.

Amen.

Reinhold Niebuhr

Acknowledgements

I wish to express my sincerest thanks to Dr Stephan Reiff-Marganiec for all his help throughout the duration of this research. His guidance, enduring patience and persistence have certainly been invaluable and without his gentle pushing this research would never have been completed. I am especially grateful for the assistance he provided towards the end of this period, when he often gave feedback at short notice.

I also wish to thank the European Union for funding the Sensoria project, which provided me with financial support for the first two years of this work until I left to work for ATX Technologies. I would also like to thank Prof. José Luiz Fiadeiro for his encouragement to complete this work whilst working at ATX.

This work was conducted with the input and support of a number of others from the Department of Computer Science in Leicester, including Laura Bocchi, Reiko Heckel and Fer-Jan de Vries, plus colleagues on the Sensoria project, namely Dr Carlo Montangero and Dr Laura Semini. Also, my fellow inhabitants of G1 (namely Dénes, Harry, Mark and João) provided a great environment for conducting this work.

Finally I would like to thank Cathy for her support as I have strived to complete this whilst still working full time. Her support has been invaluable.

Thanks!

Stephen

Contents

1 Introduction **1**
1.1 Business Processes 2
1.2 Service-Based Computing 4
1.3 Variability in the Business Domain 6
1.4 Inevitable Choice 10
1.5 Research Questions 11
1.6 Document Outline 12

2 Background and Related Work **14**
2.1 Introduction 14
2.2 Service Oriented Architecture 15
2.2.1 Related Technologies 17
2.2.2 The Promise and Limit of SOA 18
2.3 Business Process Management 20
2.3.1 Code-Based Descriptions 21
2.3.2 BPEL 22
2.3.3 Calculi-Based Descriptions 25
2.3.4 Notation-Based Descriptions 26
2.3.5 Adaptive Workflow Management 29
2.3.6 Evaluation of Workflow Specification Methods 30
2.4 Policies 31
2.4.1 Combining Policies and SOA 33
2.5 Summary 33

3 StPowla **35**
3.1 Introduction 35
3.2 Promoting Dynamicity 36
3.3 Graphical Workflow Notation 37
3.3.1 Start and End Points 38
3.3.2 Process 38
3.3.3 Task 38
3.3.4 Flows and Scopes 41
3.3.5 Operators 42
3.3.6 Workflow Execution 46
3.4 Policies 47
3.4.1 Appel 48
3.4.2 Policy Attributes 49

3.5 Tasks and Services . . . 51
3.6 The Service Level Agreement Language . . . 56
3.7 Pragmatics of the Customization . . . 57
3.8 Summary . . . 57

4 Workflow Reconfiguration 59
4.1 Introduction . . . 59
4.2 Graph Transformation . . . 61
4.3 Workflow Ontology . . . 63
4.4 Reconfiguration Functions . . . 65
4.4.1 Insert . . . 67
4.4.2 Inserting Operators . . . 74
4.4.3 Delete . . . 87
4.4.4 Fail and Abort . . . 94
4.4.5 Block . . . 96
4.5 Summary . . . 98

5 From StPowla Processes to SRML Models 101
5.1 Introduction . . . 101
5.2 SRML Foundational Concepts . . . 104
5.2.1 Transitions . . . 107
5.2.2 Business Roles: the Interactions . . . 109
5.2.3 Business Roles: the Orchestration . . . 109
5.2.4 Constraints for Service Level Agreement in a SRML Module . . . 111
5.3 Basic Control Flow Encoding . . . 113
5.3.1 Sequence . . . 116
5.3.2 Flow Junction and Flow Merge (XOR) . . . 117
5.3.3 Flow Split and Conditional Merge (AND) . . . 118
5.3.4 Strict Preference . . . 119
5.3.5 Random Choice . . . 120
5.3.6 Scope . . . 121
5.4 Advanced Control Flow Encoding . . . 123
5.4.1 Refinement Policies . . . 123
5.4.2 Reconfiguration Policies . . . 124
5.4.3 Reconfiguring the Procurement Scenario . . . 131
5.5 Summary . . . 131

6 Case Study 133
6.1 Introduction . . . 133
6.2 Scenario . . . 134
6.3 StPowla Representation . . . 136
6.4 Encoding to SRML . . . 142
6.4.1 Methodology . . . 143
6.4.2 Use Case Driven Example . . . 144
6.5 Summary . . . 149

7 Evaluation **150**
7.1 Introduction 150
7.2 Capabilities and Limitations 150
7.3 Workflow Patterns 152
7.4 Critical Assessment 153
7.4.1 Basic Patterns 156
7.4.2 Advanced Branching and Synchronisation Patterns 159
7.4.3 Multiple Instance Patterns 167
7.4.4 State-based Patterns 168
7.4.5 Cancellation and Force Completion Actions 170
7.4.6 Iteration Patterns 173
7.4.7 Termination Patterns 175
7.4.8 Trigger Patterns 177
7.5 Policy Conflict 178
7.6 Business Value 179
7.7 Summary 182

8 Conclusion **183**
8.1 Introduction 183
8.2 Reflection on Research Questions 184
8.3 Limitations and Further Research 188
8.4 Beyond SOA 190
8.5 Summary 191

Chapter 1

Introduction

If you don't like something change it; if you can't change it, change the way you think about it. (Mary Engelbreit).

The concept of us humans being able to work within rigid structures continuously is very difficult. The very notion restricts any form of creative thought, self-improvement or even innovation. Processes would become blindly followed with any hint of potential improvement ignored. Conversely, computers thrive in situations where repeatable actions must take place. Automation plays an increasing role in society, from product line management to data management to process management. Combining the computational advantage of automation and the human advantage of innovation might be considered as a holy grail of sorts - humans delegate all possible activities to computers and consume the significantly increased output volume.

However, matching automation to innovation is not straightforward. Put simply, getting a computer to do exactly as humans want is extremely complex. Natural improvements often occur as systems evolve, but to have a computer successfully execute a process under every possible environmental and contextual variable is often impossible, save the

highly unlikely event that all possible scenarios are catered for.

Processes, rather than structures, are more likely to change based on environmental and contextual variables, so our attention turns to incorporating these variables into existing processes. We do not seek to redesign processes to cater for each scenario, but instead find a way to enable processes to react to certain conditions in a way that separates the reaction from the core process.

In this thesis, we take the concept of *business processes* and consider them as a sequence of automated activities as an implementation of human design. We proceed to take such processes and incorporate variability at a level which is overlaid on the process itself. We seek to separate core functionality from variability and find a way to combine the two.

1.1 Business Processes

Business Process Management (BPM) is a term applied to the overall control of activities performed by companies in pursuit of achieving their particular goals, generally making gain by servicing a group of clients. Processes are considered "a generic factor in all organizations. They are the way things get done" [21]. One common method of managing processes is to document them in a structured way. The tool used for such a thing is known as the *workflow*, which is defined as:

"...the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules." [79]

Typically detailing how and when certain tasks are to be carried out in relation to other tasks, workflows can show even how and what information is transferred between tasks if necessary.

Workflows enable a precise definition of processes and the constraints under which they operate. This definition permits the use of computers to assist in automating the workflow in some way. Such assistance is given by a workflow system. Workflows, however, have one significant constraint: they are fixed[1]. In order to change a workflow, one must manually change it to the required degree. This is different from business process reengineering [45], which seeks to redesign business processes in order to make them more efficient, whilst still performing the same task (i.e. business process reengineering seeks to create efficient, functionally-equivalent new workflows from older ones).

The concept of change in workflows is a powerful one. Even though processes themselves may not change much, constraints around them may still change. For example, every business process is, or should be, governed by corporate rules and guidelines. Furthermore, every business is governed by state law and then by international law. A change in any of these could lead to a change in the workflow. At present, that would require some manual work to partly redesign and rewrite the workflow so that it conforms to new requirements.

Van der Aalst and Jablonski [91] identify issues and solutions of workflow change with respect to workflow management systems (WFMS). Their solution is to modify the WFMS and the workflow when change is required. At best, this still includes a manual rewrite of the workflow. At worst, it requires arduous work to be done on the WFMS in order for it to handle the updated workflow. Neither can be assumed to be "bad", but the solution is more like a work-around rather than a means to solving the problem in the first place.

In summary, without overly complex models it is difficult to capture all requirements for each workflow. Furthermore, requirements often change over time and are even different between cases, despite there being a core process. These differences can be captured in a single workflow, but not without redesigning the workflow to capture all extra requirements. This leads to even more overly-complex models that are more difficult to design, analyse and modify, especially when requirements change.

[1] by "fixed", we presume it to mean final, incapable of change by itself and static.

1.2 Service-Based Computing

One potential solution to this problem can be seen through the advent of Service Oriented Architecture (SOA). This relatively new software development paradigm works on a publish-find-bind mechanism, i.e.:

- A service author first develops, then *publishes* the service on a network, publicising its availability through some form of directory.
- A service consumer then searches for and *finds* the service they require using this directory.
- That consumer then *binds* their system to that service in order for the functionality of the service to be made available to the consumer.

Whilst at the moment industrial use of SOA is often on an intra-enterprise level, there is the prospect of many more services being released and the emergence of a competitive service marketplace, shifting the attention from service infrastructure to service management [19].

One of the key features of SOA is that it is technology-agnostic, i.e. it does not matter which technology one uses to access a service, since communication is based on an open standard. A benefit of SOA is that services are considered as computational components that can be used in a software application that is exposed to the network in which services are made available. This can include a workflow management system. As such, systems can be developed heavily based upon such services, (re-)using functionality that has already been published and having the potential to swap over services as and when necessary.

This last point is important when coupled with the concept of change in workflows. Supposing that a manual rewrite of the workflow is static, i.e. it must be done offline, then the

alternative is to make the change dynamically, i.e. whilst the workflow is executing. In order to retain operational efficiency, this change should be automated. SOA helps here in that the functionality of an activity within the workflow may already be implemented by an existing service. The challenges are now twofold:

1. How to map workflow tasks to services?
2. How to dynamically modify workflows?

If SOA can fulfil the functional requirement of the workflow, then the first challenge is to combine SOA with BPM in some way. An illustration of this can be seen in Figure 1.1, where tasks in the workflow are associated with a service in the technology domain.

Although a new problem has been introduced, it serves to lessen the impact of the second, original problem. The expected result is described clearly as follows:
"The BPM-SOA combination allows services to be used as reusable components that can be orchestrated to support the needs of dynamic business processes. The combination enables businesses to iteratively design and optimize business processes that are based on services that can be changed quickly, instead of being hard-wired. This has the potential to lead to increased agility, more transparency, lower development and maintenance costs and a better alignment between business and IT." [52]

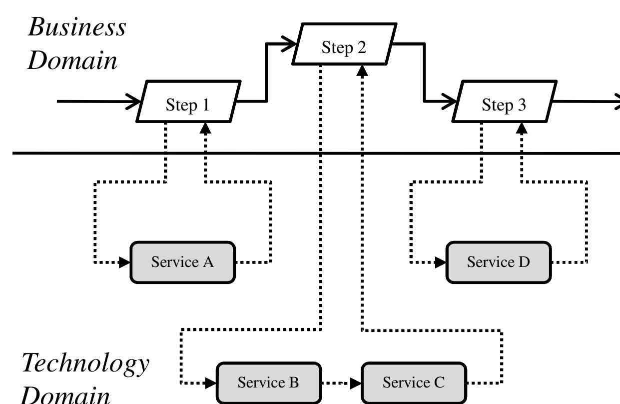


Figure 1.1: Domain separation between business and technology

Attempts to bridge the gap between the business domain and the service domain are often made by expressing business logic through composition or other technologies, but there is a distinct lack of tools which can express precise user requirements at the business level. While existing solutions tackle aspects such as functionality and sequencing of business activities, none are complete to encompass all information required at the business level [39].

1.3 Variability in the Business Domain

It has previously been identified that there exists a degree of variability within the business domain. A business process may have a standard core model, but it is possible and even likely that there will be multiple variations to this model, including the potential for changing constraints that govern it. Flexibility in workflows is thus required to handle changes at a number of different levels of abstraction, from fine-grained changes (such as a new requirement to pause at one stage) to overarching concerns (such as state law).

There is an obvious need for flexibility in the modelling formalisms for business processes. Flexibility permits the customization of a core model in order to adapt it to various requirements, and to accommodate the variability of a business domain [36]. However, since most processes are long and complex, neither manual intervention nor process termination are satisfactory solutions [43].

We consider a simple example to illustrate the point. A typical supplier business sells products to its customers and the workflow consists of the following steps[2]:

1. Supplier receives order.

2. Supplier collects items in order.

[2]We make assumptions that the client is known to the supplier and that there are no problems with stock inventory or shipping.

3. Supplier ships items to client.

4. Supplier bills client.

This workflow is a smaller part of the full workflow shown in Figure 1.2. This workflow is a description of the control flow between a number of different activities, known as tasks. These are positioned in sequence following a *Start* position and before an *End* position, which show the workflow entry and completion points. This diagram shows two participants to the workflow: firstly the supplier whose tasks are in the left “swimlane” and secondly the client whose tasks are in the right “swimlane”. The solid arrows between tasks show the control flow progression whilst the dashed double arrows between cross-swimlane tasks show a synchronisation (i.e. a point at which two tasks execute concurrently, normally as two sides of a communication). Control flow cannot continue to the next task until synchronisation has occurred (i.e. the two tasks involved in synchronisation have completed).

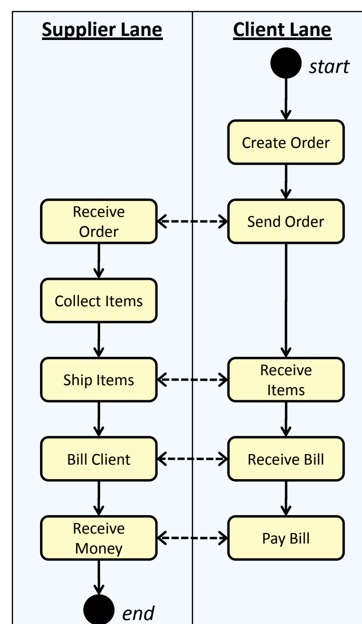


Figure 1.2: The full workflow for the supplier.

Now we consider that an order can be of small value (e.g. less than £1,000), or of significant value (e.g. £1,000 or more). Since the supplier may have to order in new stock

to handle larger orders, they may want security on orders of significant value. Therefore, they will modify the above workflow to incorporate an extra step for orders of significant values as follows:

1. Supplier receives order.
2. Supplier bills client for part of the order value.
3. Supplier collects items in order.
4. Supplier ships items to client.
5. Supplier bills clients for remainder.

The new step is straightforward, but such a modification would lead to two workflows in place to do almost the same thing (the variation of which is shown in full in Figure 1.3). This leads to needless duplication in the workflow and potential difficulties during maintenance. An alternative option would be to incorporate conditional activities inside the workflow using well-known if/else operators (see Figure 1.4). However, this will lead to rewrites of the workflow whenever a new condition is identified. If not carried out carefully, this could lead to an explosion in the size of the workflow and multiple duplications of tasks. Furthermore it could lead to difficulty in understanding the workflow, if it has become too large.

A more intuitive solution for end users is to be able to describe the business process in terms of its core model first, then its variability separately. This enables a clear focus on the core process and a separation for those aspects which can affect it. For example, we could have the workflows in Figure 1.2 together with a statement that says "*for large orders a deposit is required before the collection of items*".

A functional change to a workflow is defined as a modification to the control flow specification and this may include the insertion of a new task, the deletion of an existing task or a combination of both. Furthermore, any workflow task may be cancelled, blocked or

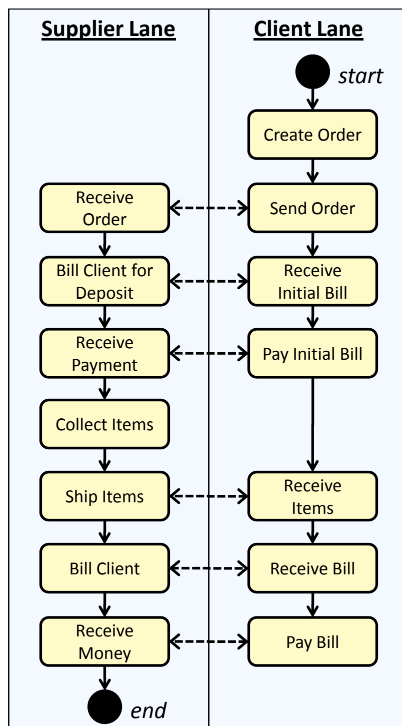


Figure 1.3: This is one possible solution to the request for a deposit when the order value is significant. There is simply a new set of tasks inserted into the workflow but there exists the requirement to manage which workflow to be executed.

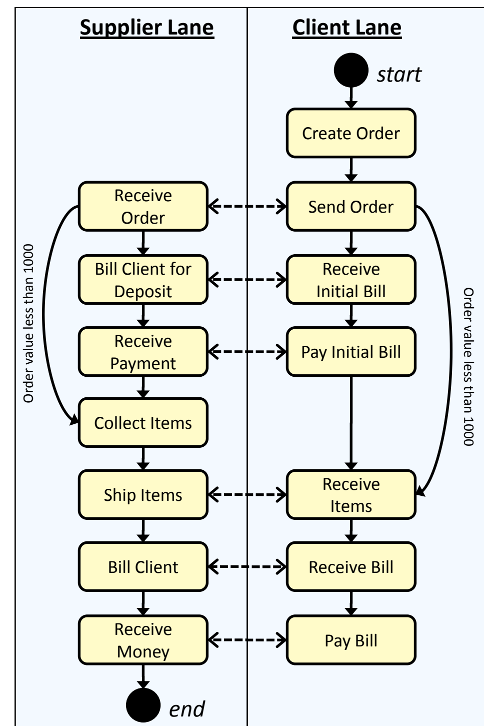


Figure 1.4: Alternative to Figure 1.3, this workflow incorporates both but uses a condition statement. For simplicity, the full if/else operation has not been described but there are two output routes to two tasks, and these are chosen depending on the given condition.

aborted without it being removed from the control flow. Such a change is known as a workflow *reconfiguration*.

A non-functional change concerns how the workflow operates, in terms of the qualities required from the services it associates with each task. Since a workflow is dependent upon services for its execution, it follows that any quality requirements must be passed on to the services being invoked. Thus, a service that does not fulfil a given quality requirement should not be selected. If each task specifies the functionality of the service it requires, a non-functional change is reflected in this specification. Such a change is known as a workflow *refinement*.

The result of these possible changes is twofold: a workflow that can change and the

services underneath that can be selected according to non-fixed criteria. Overall, the possibility is to have a workflow that is as "dynamic" as possible, i.e. almost anything can change in the workflow at runtime.

1.4 Inevitable Choice

Consider the two extremes. Workflow *A* specifies no tasks at all and the control flow is thus empty. This represents the core business process and this is accompanied by a variability specification that is external to the process but defined in association with it. This variability specification has the ability to modify all of the workflow as it is being executed, so tasks can be loaded in a particular order according to the specification.

Workflow *B* is a significantly complex workflow with every single possible condition identified and embedded into the control flow. Variability is specified separately to the workflow but since everything has already been taken into account, this specification is empty.

Clearly both of these extremes have significant disadvantages. In the first case, a lot of effort will be required in the design and development of the variability specification, especially since there is no core process on which to base any change. Variability is then designed according to other variability and the possibility of conflict between specifications arise.

In the second case where there is no requirement for variability, the old problem of redesign and rewriting appears[3]. As such, the very reason variability is allowed to be presented separate to the core process is ignored.

Thus there exists a choice for the workflow designer in terms of how much of a core process they wish to define. Returning to the supplier example above, the core process could

[3] Where a business process must change, we consider that the appropriate exercise is Business Process Reengineering, wherein a process model is potentially redesigned

vary in a number of ways. For example, the billing task at the end could be removed from this workflow and placed in another workflow. Alternatively, a variability specification could say that at the end of the process, if the client has an outstanding balance remaining, they are billed for it.

1.5 Research Questions

Having understood that business processes cannot cover every eventuality and variability is the only way to cater for this, we now identify three research aspects that we tackle in this thesis:

1. On design:
   - How to effectively design or model business processes for the business domain?
   - How to express variability over a business process?
2. On scope:
   - What kind of variability is required?
   - In what way can the business process be changed?
   - What limitations should there be?
3. On combination of business processes and variability:
   - How can we achieve this combination in a generalized way?
   - How can we combine an arbitrary number of variability items with a single business process?
   - What limits the combination in terms of capability, i.e. what can we express and what can we not express?

Several other research questions can be asked, including questions on rollback and compensation on the event of failure. Although we will briefly look into these questions inside this thesis, they are not the main focus of this research. Thus we limit our work to finding a solution for expressing workflows with variability over Service Oriented Architecture.

Thesis Statement: We desire the capability to express variability over a core business process at an abstract level for the business domain. We present an approach that provides this capability and demonstrate the scope and effect of such variability over a workflow model, where each activity is defined with requirements only and those requirements are eventually implemented by Service Oriented Architecture.

1.6 Document Outline

The following chapter discusses a number of topics that surround the subject matter of this thesis. In particular, it covers Business Process Management, Service Oriented Architecture, process variability and policies. For each, we will discuss what they are and why they are important in this context. Following this discussion, we introduce StPowla in Chapter 3. StPowla is a combination of workflows, SOA and policies, aimed at solving the problems identified.

After, we extend StPowla concepts to include the means to express variability through policy functions. Furthermore, we identify a set of functions and map each one to a graph transformation rule. In Chapter 5, we formalize StPowla by providing an encoding to SRML.

Having provided a comprehensive overview to StPowla, including its relationship to SRML, we proceed in Chapter 6 to applying StPowla concepts to a real case study, provided by an industrial partner on the Sensoria project.

In Chapter 7, we take an objective view of the material presented in this thesis and thus evaluate this work to determine adequacy, effectiveness and business value. Finally, we draw our conclusions in Chapter 8.

Chapter 2

Background and Related Work

There is nothing wrong with change, if it is in the right direction. (Winston Churchill).

2.1 Introduction

In the previous chapter, we identifed the role of workflows in business process management. We also discussed how workflows are essentially static, i.e. during execution they do not change and therefore they generally do not take into account every single possible process outcome. In this chapter, we consider existing work relating to this area, including descriptions of Business Process Management, Service Oriented Architecture and business rules.

We will also argue that no one technology currently exists to provide a suitable solution for our problem. We will identify that each of the above technologies has a role to play, but that each by itself does not offer a comprehensive solution or indeed the majority part. In addition, we will also review some approaches to introducing flexibility in workflows and provide discussions as to how they fit with our problem.

2.2 Service Oriented Architecture

Service Oriented Architecture (SOA) is a conceptual model for the design, development and deployment of software as distributed, loosely-coupled and technology-agnostic components accessible over a network. It has been referred to as the next evolution in software development [35]. Aad van Moorsel writes entertainingly about the future he foresaw for the business web [94]. He remarks that Web Services play a significant role in all (business) domains, describing them as the "Final Layer in the Internet Stack".

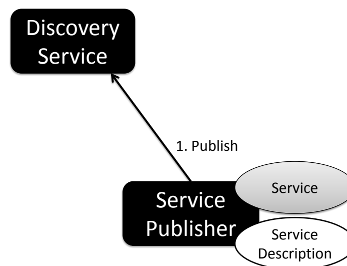
The concept of a Service is understood better as a computational unit that offers some functionality, without saying how it will do it. A suitable analogy is (ironically) a software development firm who offers a service to design, develop and deliver software applications. Clients are not often concerned with how this firm achieves this goal, but more that they do indeed achieve it.

Services are typically created and deployed according to a publish-find-bind mechanism shown in Figure 2.1. The overall benefit of SOA is the exposure of application units as self-contained and discoverable functions to the world, regardless of technology. Thus, a PHP application running on an Apache server can invoke a Java Web Service, hosted on a Tomcat server half way around the world. SOA removes the constraints of geography and technology, and encourages interoperability.

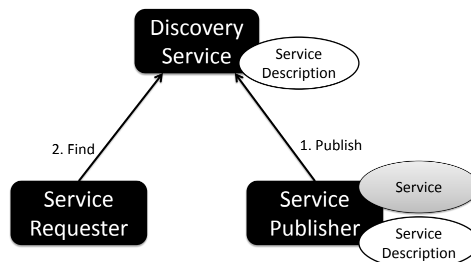
Our interest in SOA is because the workflows we will consider are at the business level, i.e. abstract from technical implementation details. As such, the functionality of the workflow must come from somewhere and SOA fits the model particularly well in the absence of real alternatives. For example, tasks can communicate with services by passing open-standard messages, whereas services can be interchangeably used throughout an entire workflow.

Our business level workflows should not be confused with lower level technical workflows. For example, since Services can be orchestrated to form a composite service, there

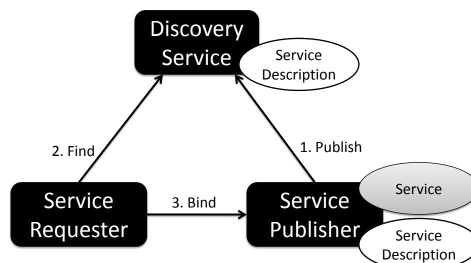
is a natural requirement for analyzing the workflow at the orchestration level. Moving towards the business level, we can also consider the workflow that defines how activities in the business workflow interact with services. So for clarity, we are concerned with workflows at the business level, and with SOA in general as it offers an implementation strategy for those workflows.



(a) Step 1: Service description published to the Discovery service.



(b) Step 2: Service requester finds service through Discovery service.



(c) Step 3: Service requester binds to the service.

Figure 2.1: The publish-find-bind mechanism of SOA.

Web Services [4] is the most common implementation of SOA and is widely used in industry[1]. The technology can be concisely described as follows:
"A Web Service is a (computational) component deployed on a Web accessible platform provided by a service provider to be discovered and invoked over the Web by a service requester." [46]

Apart from the basic requirements for a service in a SOA, a Web Service should also be describable (in terms of its available functions, their input parameters and return objects) and discoverable (i.e. information about the Web Service should be published in a directory-like system, such as UDDI [8]).

2.2.1 Related Technologies

SOA is often spoken of in relation to, and sometimes in confusion with, Grid Computing and Cloud Computing. True, these technologies may take elements of SOA, but they are not to be taken as direct implementations of SOA. The former refers mainly to an organisation of computing resources whereas the latter refers mainly to computing platform.

Grid computing can be defined as a collection of loosely-coupled, self-sufficient computational resources which connect via a network to perform collaborative, often technical, computations, usually over a significant quantity of data that would ordinarily be too much for one computer. IBM defines Grid Computing as:
"...the ability, using a set of open standards and protocols, to gain access to applications and data, processing power, storage capacity and a vast array of other computing resources over the Internet. A grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of resources distributed across multiple administrative domains based on their (resources) availability, capacity, performance,

[1]The term "E-Services" has also been used (e.g. [9, 55, 88]), mainly by industry, but usage has generally receded in preference to the phrase "Web Services".

cost and users' quality-of-service requirements." [50]

Grid Computing is different to SOA in that it focuses more on computational resources and has greater synergies with a super computer, where a key difference is that the grid components are self-sufficient and connected via a network whereas in a super computer, all resources are shared and connected via the main computational bus. Modern literature on grid computing has often described a "grid" as a virtual super computer.

Cloud Computing is defined as:
"... a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [31]

Cloud computing takes elements of SOA in that computational units inside the Cloud can be made available as services. The name "Cloud" is derived from a cloud, where people can see the external appearance, but are unable to see the internal structure or activities (if indeed there are any). Thus, these units are not seen by the end user and therefore may make use of further resources available in the Cloud in order to perform its specified purpose (see Figure 2.2). Services can be hosted inside the cloud as well as normal applications. As such, a service available in the Cloud appears no differently to a service available through other (public) means.

2.2.2 The Promise and Limit of SOA

SOA promises dynamicity. Since services are loosely-coupled, they can run continually in isolation and ignorance of others. An orchestration technology will co-ordinate the execution of services as required. Services can be continually published and found, thus

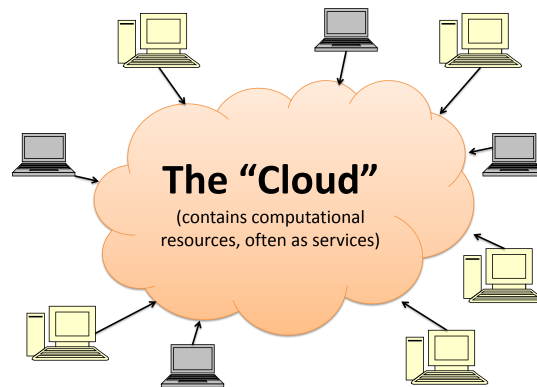


Figure 2.2: The Cloud as a platform for computing

providing the capacity for the satisfaction of almost any computing requirement. Furthermore, Web Services are composable. If a single service cannot fulfil a requirement, it may be that the requirement is a composite action, for which the broken-down atomic actions themselves may be satisfied by an existing service. In this way, a composite service can satisfy a complex (i.e. contains two or more atomic actions) requirement.

What SOA lacks is the dynamic finding of services that automatically matches service functionality to service requester, or end-user, requirements. Some inroads have been made into this field, such as ontologies and the semantic web initiative. A context-based approach for service selection has been proposed in [103, 105, 104]. One of the difficulties is to accurately describe non-functional requirements and then match those to non-functional properties of services. A suitable starting point for the latter is found in [67] and [69].

Business Process Management provides a high-level (i.e. abstract from technological implementation) to orchestration, but it does not solve the dynamic matching problem.

2.3 Business Process Management

Business Process Management (BPM) is the overall governance and monitoring of executing business processes. Business processes may be defined as structured workflows and the execution of these may be automated through a workflow management system (WFMS). Thus there are three aspects that are considered here.

We start by considering and expanding on the topic of workflows and workflow modelling. These structured activities can be defined informally (e.g. through natural language such as English) or formally. Since informally specified workflows are usually ad-hoc and lack structure, we use them only for the abstract description of processes and focus on more formal workflow specifications. Even so, there are degrees of formality that can be applied to workflows and different details that can be expressed.

Evaluation of workflow specification methods varies greatly. Since many methods are specific to a particular situation, it is difficult to compare one with another. However, a set of workflow patterns, which can be understood as design patterns for workflows, have been established in [93] as a general criteria set to which each workflow or process modelling language is compared to for determinations of its capability (i.e. how many patterns it can express) and expressibility (i.e. how well can it express).

It could be argued that modelling notations for business processes exist in their 10,000s, ranging from implementation level to the business level. The distance from implementation level reveals the amount of abstraction (i.e. the closer to the business level, the more abstraction is needed).

The value of BPM is automation of business. In the last 20 years, automation has been increasingly introduced within IT infrastructures as systems are increasingly designed to interact and integrate with others. Customer Relationship Management (CRM) systems are frequently linked with email systems and shared document systems, and new portals are created for each system to expose their functionality to new devices, such as mobile

phones. What results is that one function's completion can trigger the next function in a logical path. Such a trigger is natural from a human perspective, but slow. It traditionally requires the human to complete the original task and then chose to start the next one (assuming the next task has already been identified), potentially passing it to another person who might be in a remote location. Automation helps remove this delay. We consider that as business gets more complicated, automation plays an increasingly significant role in enabling companies to remain efficient in their activities, with regard to time, effort and cost.

In a recent white paper [64], Jasmine Noel notes that the future of enterprises is in BPM suites, therefore we consider the various BPM approaches as significant in this research. The following subsections discuss the types of workflow specification currently available, from low-level technical implementations that use a structured computer language, to formal implementations that use mathematical languages, to high level (or abstract) languages that use graphical representations. Although a description of each category is provided with specific examples, it is our aim to focus on the final category which may be considered "in the business domain" compared to the other categories.

2.3.1 Code-Based Descriptions

Code-based workflow descriptions use a recognised and/or developed computational language for expressing workflows. In terms of Web Services, the Business Process Execution Language (BPEL) is one of the most widely used process languages, even though it was designed for service composition. Other solutions include WSCL [7] and WSCI [6]. The main problem with these solutions is that they all represent implementation-level approaches to describing service interaction[2]. While they perform their respective roles as required, they are not able to express high-level user requirements. They are suitable to model processes within which details about each involved service are already known. A

[2] We note that BPEL engines exist from different vendors, but the BPEL standard is common to all

comparison of ebXML, XLANG, WSFL and BPML can be found in [66].

Modelling generic processes requires a more holistic view. YAWL [92] is a workflow language that extends Petri nets to provide a powerful formal language with defined syntax and semantics. YAWL was created as a reference language for evaluating workflow implementations and it has an interesting capability known as Worklets [3]. These worklets are “small, self contained, complete workflow processes which handle specific tasks (actions) in larger, composite processes (activities)”. YAWL enables a prescribed workflow to call, in a service oriented way, a worklet for each activity in the workflow, allowing the use of contextual information to substitute standard work items. This approach certainly has a number of merits, not least that tasks are dynamically determined based on information available. However, each worklet is still designed (not selected) statically and this is one area we are trying to improve upon.

Further languages include SMAWL [82], GAT [62], XRL (initial blueprints in [56]) and WSFL [57]. These solutions are considered better in terms of describing processes since they abstract away composition details that would be included in those solutions previously discussed. Although they describe processes, they are unable to define high-level requirements for activities or events that occur in the workflow.

2.3.2 BPEL

We reserve a specific discussion on the Business Process Execution Language (BPEL) [32] as it is widely acknowledged as the leading Web Service orchestration language. Derived from IBM’s WSFL [57] and Microsoft’s XLANG [86], the use of BPEL has grown significantly to having implementations from a number of vendors, including Active Endpoints (Active VOS), ASF (Apache ODE), Microsoft (BizTalk), Magic Software Enterprises (iBolt), JBoss (JBPM), Oracle (Open ESB and BPEL Process Manager), OW2 (OW2 Orchestra), Parasoft (BPEL Maestro), Petals Link (Petals BPEL Engine), SAP AG

(SAP Exchange Infrastructure), OpenLink Software (Virtuoso Universal Server) and IBM (WebSphere).

As a business process modelling language, BPEL has been evaluated against the set of workflow patterns [101], where the authors positioned Web Service composition languages as variants of traditional process modelling languages [90]. The conclusion was that whilst BPEL performed favourably against BPML, WSCI and XPDL (which themselves only support about half the tested workflow patterns), there were still a number of patterns that it did not support and a few negative comments such as "*it lacks orthogonality*" and "*the semantics are not always clear*".

Yet BPEL still plays an important role in identifying the scope of our work. Despite its name including the term "business", it is not business as we understand it. More precisely, for BPEL the term "business" refers mainly to functionality sequence at the orchestration level, including information on binding and sending/receiving variables. Already this is too detailed information for our domain. Thus, our interpretation of "business" is more at the user level, where the role of a business analyst is to define requirements for applications rather than implementing them. Clearly though, there exists a gap between our interpretation of "business" and BPEL's, but that remains out of scope for this work. For ease of understanding, we assume there exists a means to map our business domain to more technical layers such as BPEL.

Combining Variability with BPEL

At the time of writing, a low number of proposals for adding variability to business processes have been offered by the research community. Typically, their scope is constrained to the orchestration level (i.e. BPEL) and even then the amount of variability that is possible is still limited. A common approach is to provide a variation to Aspect Oriented Programming, where cross-cutting concerns can be defined independently of the core system. Such capability is useful for logging, authentication, etc., but provides limited

usefulness for our problem. In this section, we provide a brief discussion of some of those proposals and discuss how they compare against the problems we have identified.

In Courbis and Finkelstein's position paper [23], the authors propose the use of dynamic aspects, à la Aspect Oriented Programming [54], to introduce variability at the orchestration level. Indeed their paper shares some assumptions with this thesis, namely that Web Services can be selected at runtime rather than design time and that the BPEL engine should be "minimal but easy to configure" (we can consider the BPEL engine concept is similar to the WFMS concept). Their approach is to introduce a BPEL interpreter, which reads normal BPEL scripts and has the ability to execute a limited set of instructions capable of, for example, modifying the engine's behaviour through so-called "hot-fixes". We consider this a potential solution for our problem, yet it is not at the business level (business analysts are not expected to know how to program in Java). One could also consider that this approach is not fully in keeping with the service oriented nature of the overall context.

Charfi and Mezini present AO4BPEL, a fully aspect-oriented extension to BPEL [20]. The authors argue, as we would, that process oriented languages like BPEL lack support for dynamic adaptation of the composition even though that may be a frequent requirement (e.g. in the event of unavailability or variations in critical QoS requirements). Furthermore, business rules are identified as cross-cutting several BPEL specifications and are not well modularized in BPEL specifications. As before, advice could be the additional execution of a BPEL activity. AO4BPEL supports before, after and around advice types but it is not clear the extent to which the business process can be changed. For example, it does not appear possible to enforce a change on a change, i.e. modify a process specifically at a point where it had been previously modified.

Karastoyanova and Leymann combine BPEL with WS-Policy to achieve adaptive orchestration logic to boost flexibility and promote reuse [53]. WS-Policy is used to define the aspects and their functionality, which is limited to Web Service operations. Again, advice

is either before, after or around a BPEL activity. We note there is no direct support for "instead of" the BPEL activity. The authors recognize the problem of compensating for dynamically introduced activities and thus cannot model situations with them. The issue extends further than compensation as with the previous solution, it limits the capability of this solution to affect changes that were already dynamically introduced.

Rosenberg and Dustdar describe the relatively straightforward integration of business rules with BPEL [77]. Using a business rules engine alongside an orchestration engine, the execution of the BPEL script is always subject to what the business rules engine determines. The process involves interceptors for each incoming and outgoing BPEL Web Service call and, as required, calling the business rules engine (via a transformation process) to enforce any necessary changes. The main difference with this work to ours is the focus on BPEL, although the conceptual difference is that only Web Service calls can be changed at runtime, whereas we are looking to introduce dynamic change in the process itself, i.e. modifying the BPEL script.

In summary, the problem we identified is a top-down issue: we are looking for users to be able to express what they need and build a solution around it. Thus, the solution starts at the high level business workflow. The variability solutions proposed work at a lower level and might be considered more bottom-up approaches, where the technical implementation decides what users can achieve. In the end, a combination of approaches may well offer a rounded, if not comprehensive solution. In the meantime, we continue to determine an adequate solution for our top-down problem.

2.3.3 Calculi-Based Descriptions

A sister approach to the code-based descriptions, process calculi offer a formal, mathematical method in which to express processes. Whilst not including enough information for actual composition, this approach allows one to model a (composition) process so that

it can be verified and reasoned over (i.e. the semantics are well-defined).

Semantics are required for this formality and process calculi such as Petri Nets (already used for YAWL), π-calculus and CCS all possess structured operational semantics. Examples of this approach are described in [44] and [33]. However, process modelling in a service oriented context is not straightforward. Loose coupling of services, communication latency and open-endedness (services appearing and disappearing unannounced, geographically-responsible changes in service quality, workload distribution between semantically equivalent services, etc.) are key issues [16]. We are not concerned so much with loose coupling or communication latency, but we are concerned with open-endedness in the sense that new activities could be introduced at any time.

In the case of BPEL, a transformation to Petri Nets has been proposed [48], with the full semantics defined in [80].

2.3.4 Notation-Based Descriptions

Graphical notations generally provide an intuitive method in which analysts can define processes by a sequence of activities or tasks. As with code-based approaches, there are a number of vendor-specific solutions, e.g. IBM's WebSphere Business Modeller[3]. However, they conveniently abstract away more implementation level details which make them more suitable for use in the business domain. Indeed, business analysts have long been using flow charts to define processes, so a similar modelling notation that is adaptable to SOA is desirable.

The Business Process Modeling Notation (BPMN) [65] is a widely-accepted graphical notation for the specification of workflows representing business processes. A core BPMN workflow may consist of the following elements:

[3]http://www-306.ibm.com/software/integration/wbimodeler/

- Flow Objects: Events, Activities and Gateways;
- Connecting Objects: Sequence Flow, Message Flow and Association;
- Swimlanes: Pool and Lanes within a Pool;
- Artifacts: Data Objects, Text Annotations and Groups.

A BPMN workflow starts with a Pool, in which the workflow is described. This Pool may be divided into swimlanes, with one swimlane per entity participating in the workflow. Each activity specific to an entity appears in only their respective swimlane. Events and Activities are self-explanatory items within the workflow whilst Gateways are decision points, where the next item in the workflow differs depending on a condition. Flow Objects are connected by sequence flows and message flows provide a means of expressing data input into a flow object. One flow object may be associated to another. Data objects provide a description of the requirement or output of an activity. Text annotations are informal notes added to the workflow and groups are collections of elements within a given boundary (e.g. if the boundary is from the start event to the end event, the group contains all elements in the workflow).

Figure 2.3 shows an example BPMN workflow for the earlier supplier example.

BPMN itself is a specification, rather than an implementation. The BPMN website[4] notes that there are, as of 19 April 2011, 76 implementations and 4 planned implementations.

One significant benefit of BPMN is that it maps to BPEL specifications [68, 98]. Furthermore, it has been argued that BPMN provides a user friendly interface to lower level BPEL code, though some debate still remains over conceptual mismatches between the two languages [71].

BPMN is an example of a static workflow specification method. Returning to the example given in the opening chapter, if the author wished to change the workflow, they would

[4]www.bpmn.org

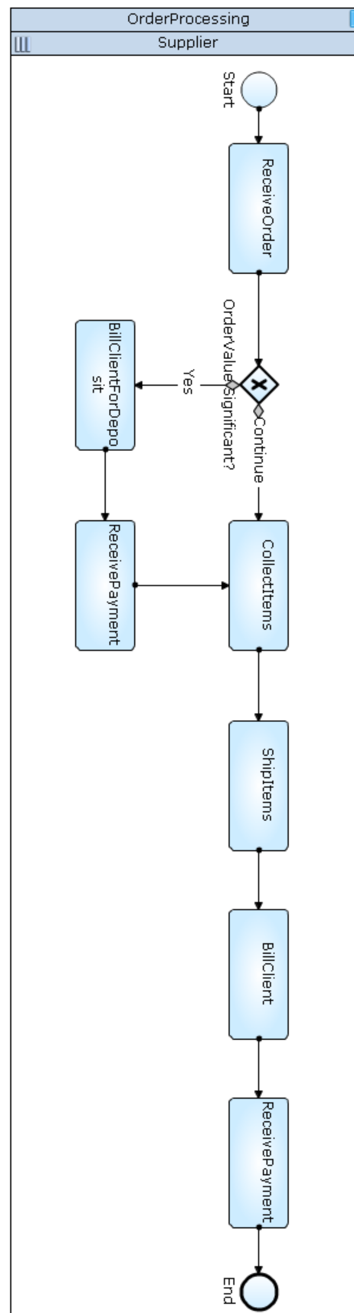


Figure 2.3: An example BPMN specification.

have to rewrite it to the required degree. This could lead to a small number of vastly complex models or a larger quantity of models that include a lot of duplication. The BPMN specification also states that it was "*constrained to support only the concepts of modeling that are applicable to business processes*", thus not supporting organisational structures and resources, functional breakdowns, data and informational models, strategy and business rules. Such extra concerns are those that can ultimately affect the design and

implementation of a workflow.

Alternatively, another notation-based description is the Activity Diagram component of the Unified Modelling Language (UML) [51]. This diagram expresses the dynamic behaviour of systems in a clear and intuitive way. However, UML often suffers from a lack of clear semantics, though the syntax is clear. Further research goes into adding semantics for UML (e.g. [26, 27, 28, 83, 84, 102]). Also, while the syntax of the diagram is intuitive, it is sometimes unclear as to what a particular layout would do. For example, in Figure 2.4, it is unclear when D is executed. Does it occur after B and C, or once after either? This problem arises when B and C take different amounts of time to complete.

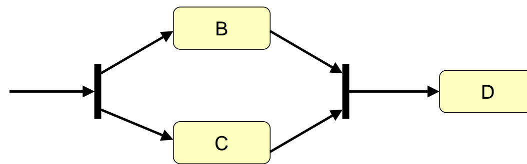


Figure 2.4: UML fork and join example

2.3.5 Adaptive Workflow Management

Having considered BPM together with some of its implementations, we also consider current efforts to permit variability in the workflow.

AGENTWORK [61] is an adaptive workflow system that supports predictive and reactive rule-based adaptation subject to failures in the process execution. *Predictive* implies that the system must guess that there will be a failure based on current information whereas reactive adaptation occurs when predictive adaptation is not possible (i.e. based on current tests, etc.). Event-Condition-Action rules are used to describe the required change. Possible changes include adding activities (in sequence or parallel), dropping activities and adding loops.

$\text{ADEPT}_{\text{flex}}$ [72] is an extension to ADEPT, a graph-based workflow modelling system,

that permits dynamic changes to the control flow using a minimal set of operations (inserting/deleting activities, skipping activities, passing over activities and changing activity sequences). The changes are available as functions on the graph, suggesting strong coupling between the core workflow system and the component that permits flexibility. It is unclear though how a business user would make use of the flexibility functions and how such functions would be available to one or more workflows (i.e. if the rules are tightly coupled with a single workflow).

Pesic and van der Aalst propose the use of ConDec [70] as a declarative approach to specifying workflow variability. Linear Temporal Logic (LTL) is the means used to define policies or business rules that define the change over the core process model, where each LTL clause defines a relation between workflow tasks. Such clauses can be substituted or rewritten with other clauses in order to effect the change desired. Although this approach appears to fit our situation well, we note that we can only modify control flow whereas although we certainly desire this capability, but we also need an approach that can offer the capability to go further than just the control flow, e.g. to data flows, error flows, etc.

2.3.6 Evaluation of Workflow Specification Methods

Notation-based methods generally lack the ability of expressing any information that is not graphical, e.g. "*this task should be completed in x time*" or similar. Therefore we require a particular capability of expressing this further information, down to the subprocess (i.e. task or activity) level. Furthermore, notations are static approaches to workflow specification, i.e. one cannot change the workflow without redesigning it.

UML Activity Diagrams and BPMN have previously been evaluated as a means for workflow specification [25, 99]. Both come up short against workflow patterns but, and more importantly for this research, neither support refinement or reconfiguration of a workflow at runtime.

Calculi-based methods are good for analysis, yet lack real usability unless a notation can be defined on top of the formal calculi. They too lack the capability for specifying non-functional requirements of a process and they also lack the capability for dynamic change.

Code-based methods are good for users who have a fine understanding of the language and concepts in use, but they are not suitable, in general, for business analysts and managers. The one exception is that BPMN maps to BPEL processes, but even that has some limitations as described before.

2.4 Policies

Policies are found in a number of different forms inside companies and organisations already today. In the United Kingdom, companies are generally required to have defined, amongst other documents, a Health and Safety Policy and an Equal Opportunities Policy. The former describes how the company will ensure their employees (and sometimes guests) will be protected from any health and/or safety risks. The latter describes how the company will treat employees and potential employees of different skin colour, race and/or faith, to name only a few variables.

Such documents are rarely static - they change as the company, industry and culture changes. For example, for organisations that work with children, it is a UK requirement to review the Child Protection Policy annually.

In computing, policies are tools described as “information which can be used to modify the behaviour of a system” [59], without the need for re-compiling or re-deploying. In essence, a system’s behaviour reacts to, or is constrained by policies. Policies are typically loosely coupled with the system they interact with. The system can run independently from policies with any resulting actions considered as *default* actions. These could arise from standard system behaviour or from executing workflows.

One may consider the relation between policies and business rules. The latter can be defined as a statement that defines and constrains some aspect of the business [42]. We can easily consider that policies and business rules are almost synonymous from a conceptual viewpoint and since there are different types of business rule, there must also be different types of policy.

A reactive policy is normally associated to an event. For example, if a staff member falls down they should see the designated first aider on duty, or committing data to a database under certain conditions may trigger an update elsewhere. A constraint policy provides boundaries that a process must operate within. For example, "*staff must work onsite between 09h00 and 17h30 Monday to Friday*", or "*batch processes should start after 21h00 and complete before 08h00 the next day*".

In order to maintain an effective IT environment, computer systems must adapt to changing business requirements. Policies have proven useful in evolving computing environments since they allow for rapid modification of system behaviour without modifying the state of the system's codeset, providing that the system has already been developed with support for such policies.

We refine the definition of policy from [73] in the context of Web Service management as:

> A high level statement as to how business requirements should be processed in the management system.

Policies can be defined from different viewpoints. For example, an organisation may have general usage policies over software whereas project teams may have further policies specific to them, and even further an individual who has their own policies.

There are at least two particular types of policy: Event-Condition-Action (ECA) rules and goals (ECAs without triggering events). Each policy encodes information about a

particular business activity.

2.4.1 Combining Policies and SOA

Policies may be applied to a variety of areas within SOA. They are currently used for access control and to express reactive functionality. However, they have yet to be applied to a business management framework.

In [85], one of the few occurrences of combining SOA and policies is proposed. The authors argue the introduction of composed services using coordination policies from the WS-Policy Framework [96]. Despite the combination, the results do not help us solve our problem.

In our search for dynamic workflows, the combination of policies, SOA and BPM leads us to the ability to define workflows as *base* functionality, which varies according to one or more policies, with the technical implementation provided by loosely-coupled services.

Our work is aimed at developing a business policy framework for the management of Web Services in the business domain. The use of policies that we propose is orthogonal to a graphical business modelling language. Each activity within a workflow represents a unit of business activity that contributes to the satisfaction of the wider business goal. Tasks are subjected to external policy inputs, or global policies.

2.5 Summary

In this section we have described three individual technologies: BPM, SOA and policies. The first two have very active research communities and high industry adoption rates. The last is also a subject of active research, but to lesser extent and this can be explained by the number of different ways in which policies can be used.

We placed a specific emphasis on policies providing variable behaviour on its subject, without removing base behaviour from the original subject. We also emphasized BPM in the context of workflow definition in the business domain, designed specifically for non-technical people to use. The technical implementation is (potentially) given through SOA.

In the next section, we will see in more detail how these three ingredients combine to form a solution for dynamic workflows.

Chapter 3

StPowla

Our only security is our ability to change. (John Lilly).

3.1 Introduction

In the previous chapter, we identified three components in Business Process Management, Policies and Service Oriented Architecture (SOA), each of which have little relevance on their own but combined they could address our problem of the need for dynamic workflows.

In this chapter, we take these three ingredients and combine them into a recipe, which we call the Service-Targeted, Policy-oriented Workflow Language, or simply StPowla. We will see how a simple workflow language is used to provide the specification of the core workflow. Policies can be applied to provide variability to the core model. StPowla integrates three main ingredients: a graphical workflow notation, a policy language, and SOA. The last ingredient provides the underlying functionality required by the workflow.

We address the integration of business processes with SOA at a high (that is close to the

business goals) level of abstraction. The reason for this is that, often, end users defining goals do not have deep technical knowledge and, therefore, it is necessary to abstract away lower level technical information. It is helpful to consider that any of the languages used with StPowla, e.g. the policies or workflows, are compiled into XML notations abiding Web Service standards. In fact this compliance to existing standards is key to interoperability at an implementation level.

3.2 Promoting Dynamicity

StPowla is aimed towards the separation of concerns between core functionality and process variability. The dynamicity required of processes is contained within variability specifications, which is defined separately to the core functionality. In fact, StPowla addresses the need for dynamicity at three distinct levels:

At implementation: the use of SOA to implement tasks offers immediate dynamicity with respect to which services are selected to implement each task. Since an assumed runtime engine matches services to tasks, the overall workflow is not bound to the same services for each execution[1].

At runtime: although the core process can run unaltered from its design, the use of policies can ultimately modify behaviour depending on contextual or environmental information. StPowla's core design intends this as the key result.

At development: the use of policies orthogonally to workflows means that core functionality and variability specifications can evolve independently of one another. The result of combining the two is a very dynamic development structure, i.e. the combination of previously isolated workflows and policies can lead to completely new workflows.

[1]We can assume of course that the reuse of discovered services may be beneficial but this is not the point we are trying to convey.

3.3 Graphical Workflow Notation

The first ingredient of StPowla is a graphical workflow notation that allows the definition of a workflow at a level abstract from technical details, i.e. at the level appropriate for business analysts (who are the users targeted for StPowla). The notation simply defines the core process in terms of sequential, parallel and decision-based composition of units of functionality that we identify as *tasks*.

Although we will use a single notation for the remainder of this document, it should be noted that the notation itself is not important. Other notations (e.g. UML [26], BPMN [65]) can be adapted to function as required. This notation, first published in [38], is used due to its simplicity, effectiveness and level of abstraction; three characteristics that make it ideal for StPowla.

Throughout this section, we will describe the workflow from the perspective of the workflow designer. The full graphical picture of the workflow is known as the task map or the workflow model.

In addition to graphical syntax, we describe a workflow with the following grammar as a means to providing a clear syntax:

WF	::=	$start; P; end$	root process
P	::=	T	simple task
	$\mid$	$P; P$	sequence
	$\mid$	$\lambda^{?}P : P$	condition and simple (XOR) join
	$\mid$	$FJ(m, \{P, \mathcal{B}\}, ..., \{P, \mathcal{B}\})$	split and complex (AND) join
	$\mid$	$SP(T, ..., T)$	strict preference
	$\mid$	$RC(T, ..., T)$	random choice

3.3.1 Start and End Points

A workflow must begin with a single Start point, denoted by a black circle with the *Start* label to the side. It is terminated at a single End point, again denoted by a black circle with the *End* label to the side (Figure 3.1). Since it is commonplace in processes to define multiple end points, this is allowed informally whereas actually all paths that lead to an end point will converge to a single path leading to the single end point. The requirement of a single end point provides precision in the workflow definition process (e.g. use of complementary operators such as Flow Split and Conditional Merge). Between the Start and End point exists a process.

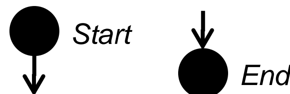


Figure 3.1: Workflow Start/End points

3.3.2 Process

A process is a sequence of workflow items between two points. Such a sequence is defined according to need, but can be a simple task, a sequence of two processes, or an operator on the control flow. Informally, we could include either or both the Start and End points, with the latter case meaning that the whole workflow is a process.

3.3.3 Task

A task is a basic (atomic) activity performed inside a workflow. Often referred to as a work item or activity, the task is the main foundational building block on which all workflows are based. A task is drawn simply as a curved-corner rectangle, with a unique identifier shown inside the rectangle, which is its label.

A task allows the designer to specify a particular activity at a specific position inside the workflow. As such, a task will include at least:

Control Flow Input: the incoming link that connects the task to earlier parts of the workflow. The point of control moves along given paths inside the workflow and when it reaches a task, the task is executed.

Control Flow Output: after the task has completed, control flow moves onto the control flow output path to the next item inside the workflow.

Functional Requirements: each task must have associated to it a set of requirements that defines exactly what the task will do. It is assumed that a task is mapped to a Service at runtime with no manual intervention from the end user. These requirements do not assume a 1:1 mapping from Task to Service, but that a separate system can match the requirements of the Task to one Service, be it atomic or composite.

In addition to these mandatory features, a task may also have some optional features:

Data Flow Input/Output: shows how data is mapped from one task to another task, when there exists a situation whereby a task outputs some data which is to be used as input to another.

Non-functional Requirements: adding to the functional requirements, these define quality requirements of the task. Specifically, they define how well the underlying service should carry out the required functionality.

External Input: a task may be subject to other variables, such as external policies. Although again not mandatory, the notation at least allows the end user to explicitly specify these external variables.

Abort/Failure Output: these are additional control flow outputs that are followed if the task is aborted or it fails. Should one of these outputs not exist and yet the event still occurs, the normal output is followed.

The image of a task is shown in Figure 3.2. Although a task is atomic, in the sense that it is defined by the end user as a single activity, the end user can specify a composite task (Figure 3.3). The composite task essentially represents a path inside the workflow between two defined points and all items between them. It could easily be understood as a process, a scope or a workflow that exists inside a workflow[2]. For example, Figure 3.4 shows such an arrangement.

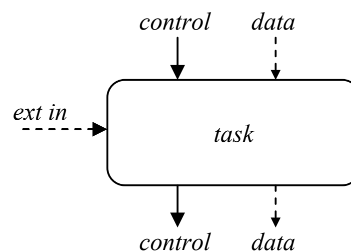


Figure 3.2: An atomic task with its input/output flows

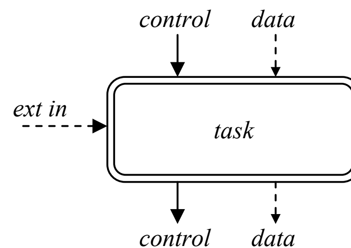
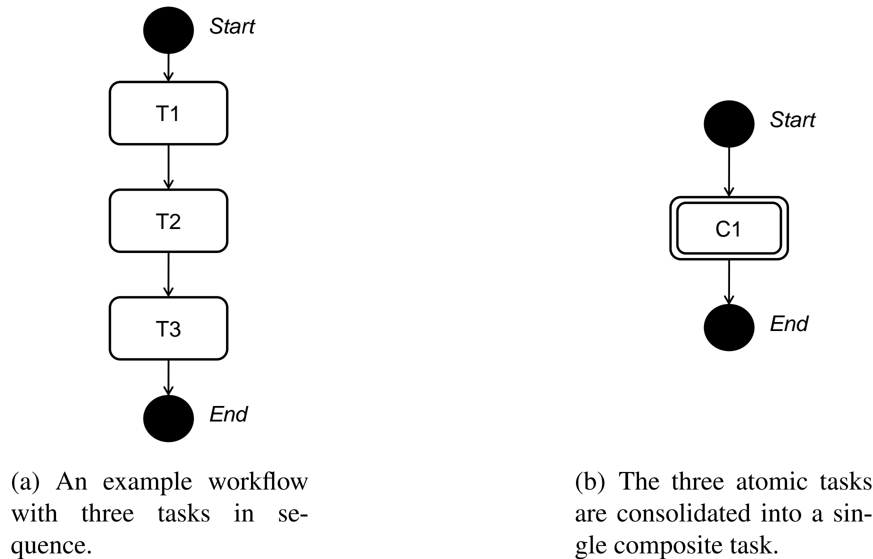


Figure 3.3: A composite task with its input/output flows

Typically it would be expected that simple workflows would be specified only in terms of atomic tasks. However, more complex workflows may include multiple composite tasks that are defined separately to the main workflow. This may involve the re-drawing of the graphical workflow model to show multiple atomic tasks as a smaller number of composite tasks. For example, a composite task could represent a fully independent workflow from a "master" workflow. A task in the master workflow may have the requirement to execute the separate workflow and this task is defined as a composite task. The exact definition of the content of a composite task is completed outside of the graphical notation.

[2]this method is typically used to bring clarity to workflow models.



(a) An example workflow with three tasks in sequence.

(b) The three atomic tasks are consolidated into a single composite task.

Figure 3.4: Example of how atomic tasks can be rewritten as a single composite task.

3.3.4 Flows and Scopes

A flow is a sequence of items (tasks or operators) in the workflow. A route is a flow that begins at the start point and finishes at the end point. A flow can either be a control flow or a data flow, although the latter is not considered in this work. Each task must be connected to the workflow by a control flow, i.e. it must be in a route. Otherwise, the task is redundant as there is no way to invoke it.

A flow may be active, where it contains an executing item, or non-active, where it does not contain an executing item.

A scope is a sequence of items in a workflow, including control flow paths, operators and tasks. It is defined by the first and last items, including all items between those two. A simple requirement therefore is that the start item must be executed before the end item in the control flow. It may not necessarily follow that the start item appears before the end item since loops may occur.

Scopes may be identified in the graphical model, simply by marking out the sequence of items through a dashed box. A scope is uniquely identified through an ID.

3.3.5 Operators

In addition to tasks and flows, which can express simple sequencing, operators are defined as functions on control flows. These further enable a business to accurately model their business process, e.g. by allowing concurrency. Throughout these descriptions, we use T to denote the set of workflow tasks.

Operators each have control flow inputs, i.e. arrows which represent the routes through which control is passed to the operator, and control flow outputs, i.e. arrows which represent the routes through which control will be passed to the next workflow item.

Sequence: this function permits the sequential execution of two workflow processes, with process $P1$ executed first and, on completion, process $P2$ executes. This is depicted in Figure 3.5. Although not shown, $P1$ becomes active through a single control flow input. Once active, $P1$ executes and, on completion, passes control flow through the arrow to $P2$, which proceeds to execute and then pass on control flow through a single output to the next workflow item.

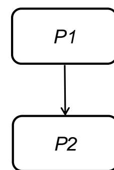


Figure 3.5: Sequence operator

Flow Split: this is an n-ary function $FS : in \rightarrow OUT$, where in is a control flow input and OUT is a set of control flow outputs. In Figure 3.6, the operator is pictured with one input and five output flows ($f_{x.1}$ to $f_{x.5}$). Of the output flows, the operator specifies that each shall become active, i.e. the current control flow position is multiplied over each of the flows and each one is executed independently and concurrently. For example, in a typical customer-supplier-warehouse example, a product dispatch may involve simultaneously notifying the customer of the dispatch whilst ordering a stock replacement. Both

tasks are executed independently.

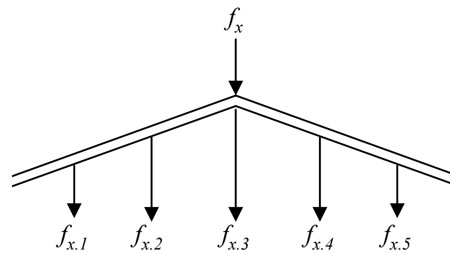


Figure 3.6: Flow Split operator

Conditional Merge: this takes a set of active incoming control flows and, subject to business-defined constraints, merges them with synchronisation to a single output flow. A solid circle at the end of the incoming flow indicates that the flow must complete (i.e. control flow must reach the conditional merge operator on this control flow) for synchronisation; this is also known as the control flow being mandatory. A clear circle indicates that this flow is optional. The number in the diamond indicates the number of control flows that should complete (i.e the last task in that flow must complete execution), irrespective of if they are mandatory or optional. As such, the number n must be greater than or equal to the number of mandatory flows. In Figure 3.7, four control flow inputs are shown. The first input $f_{x.1}$ is mandatory (i.e. it must complete) and the rest ($f_{x.2}$, $f_{x.3}$, $f_{x.4}$) are optional. The operator completes when control flow input $f_{x.1}$ completes and $n-1$ of the remaining control flow inputs complete.

For example, when looking for airline ticket quotes, one might request quotes from three suppliers, including the preferred supplier. Before booking, we might say that we must have a quote from the preferred travel agency, plus another travel agency (the selection of which is unimportant to us). Thus there is one mandatory flow and a minimum of two flows to complete before proceeding. In Figure 3.7, there are four incoming control flows (potentially representative of four travel agencies), of which the first is mandatory (i.e. the preferred travel agency) and the rest are optional.

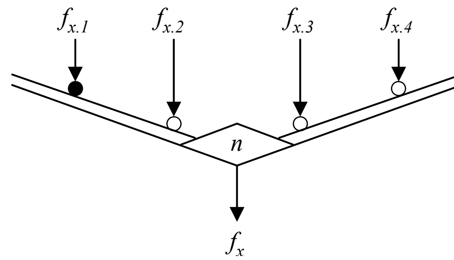


Figure 3.7: Conditional Merge operator

Flow Junction: this diverts the control flow down one of two possible output flows according to a binary test. If the result to the test is positive, i.e. *true*, control flow passes to the left hand flow and the right hand flow otherwise. The test can include any attribute or variable accessible by the operator, within the workflow, from an external input or from a global variable.

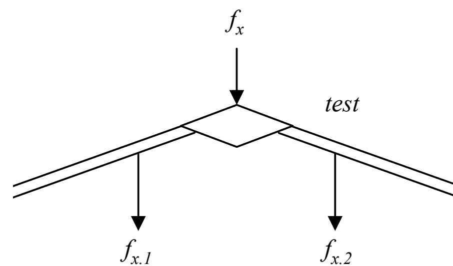


Figure 3.8: Flow Junction operator

Strict Preference: this operator attempts to execute a series of tasks in a defined order, progressing when one of the tasks is completed. The function is written $SP : in \rightarrow OUT$, where OUT is one of a set of possible output flows. We describe Strict Preference as a set $PT \cup \{error\}$. PT is an ordered set of pairs in the form (t, n), where $t \in T$ represents a task and $n \in \mathbb{N}$ is a unique priority in the range $1 \leq n \leq |PT|$, such that the task with highest priority is attempted first. In the case where no task can be completed (due to a timeout or other issue), the *error* output flow is selected.

Each task in the operator specifies its own output flow which is followed when its parent task is completed. Thus after the operator, there is one active output flow plus a number of inactive flows.

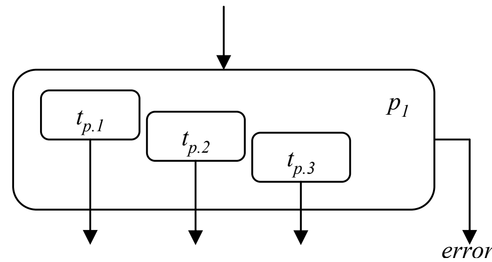


Figure 3.9: Strict Preference operator

Random Choice: this is similar to preference, but without priorities attached to included tasks. It is described simply as $CT \subseteq T$, where CT is the set of tasks included specifically in the operator. When control reaches this operator, all tasks may be attempted simultaneously. When a first task reaches a commit stage, then all others are cancelled (assuming the tasks have a cancellation function or else their results are discarded). After completion of a task, control flow passes to the flow connected with this task. Graphically, the difference with the preference operator is that all included tasks are lined up on the same plane.

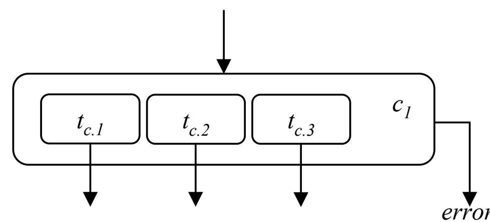


Figure 3.10: Random Choice operator

Flow Merge: this is a unary operator in that it takes a set of control flow inputs and converges them into a single output control flow. In order to preserve synchronisation (i.e. ensuring we have execution concurrency only when desired), only one flow of the incoming set must be active, with all others inactive. This may be the result of a prior Flow Junction, Strict Preference or Random Choice operator. If more than one flow is active, then the Conditional Merge operator must be used. Thus the flow merge operator can be viewed as a “simple merge”, or non-synchronising merge.

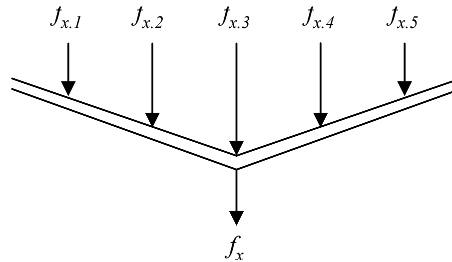


Figure 3.11: Flow Merge operator

3.3.6 Workflow Execution

So far we have described workflow representation through the written, formal syntax and the graphical syntax. Next, we must consider workflow execution.

The critical element of workflows that enable them to be used in StPowla is the set of events through which integration with the rest of StPowla is possible. The events are described in Table 3.1[3].

Level	Event Name	When Triggered
Workflow	On Workflow Start	The workflow is invoked.
	On Workflow End	The workflow control flow reaches the End point.
	On Workflow Error	A task within the workflow ends in an error and there is no specific error handling mechanism defined to handle it.
Task	On Task Entry	Control flow reaches the task.
	On Task End	Task processing completes.
	On Task Error	The task processing ends with an error.
	On Task Abort	The task's execution is externally (i.e. from outside the task specification) aborted.
Service	On Service Start	The service is invoked, following binding.
	On Service End	The service responds with an acceptable value, if any.
	On Service Error	The service responds with an error response or a timeout occurs.

Table 3.1: Workflow events used by StPowla

[3]We include service-based events in recognition of the use of SOA in completing a task. These events are triggered inside a task though, which as black boxes are more difficult to observe at the full workflow level.

3.4 Policies

The finer details of the business process (i.e. the variability) are expressed by policies. These can define functional and non-functional requirements of a task *execution*. Furthermore, they can provide *overarching business constraints*, that is a set of rules specified at a global, enterprize or project level and applicable to the whole workflow. The last part that policies might play is concerned with suggesting resolutions to particular problems in the process execution (e.g. "*do not use a service from X, rather use an equivalent service from Y*").

So, the second ingredient is a *policy language*: here we use Appel (Accent Project Policy Language) [75]. Though developed in the context of telecommunications, Appel is a general language for expressing policies in a variety of application domains. It was conceived with a clear separation between the *core* language and its specialization for concrete *domains*, which turns out very useful for our purposes.

The third notable characteristic of StPowla is that it is targeted to SOAs. Its users, though ignorant of implementation details, should be aware that the business is ultimately carried out by *services*, i.e. computational entities that are characterized by two series of parameters: the *invocation* parameters (related to their functionalities), and the *Service Level Agreement* (SLA) parameters. Stakeholders can adapt the core workflow by modifying these agreements.

Service invocation is local to task execution, i.e. a service is invoked to satisfy the requirements of a task, not to satisfy some overarching business constraint or generic instruction.

3.4.1 Appel

Appel is a policy language designed for end-users: its style is close to natural language, permitting ordinary users to formulate and understand policies readily. Appel's formal semantics [60] underpin integration with workflows.

In Appel a policy consists of a number of *rules*, grouped using a number of operators (**seq**uential, **par**allel, **g**uarded and **u**nguarded choice). A policy rule is a variant of an ECA (event-condition-action) rule, consisting of an optional *trigger*, an optional *condition*, and an *action*. To help the user, a wizard has been presented to formulate policies [87]. The applicability of a rule depends on whether its trigger has occurred and whether its conditions are satisfied.

A condition expresses properties of the state and of the trigger parameters. Conditions may be combined with **and**, **or** and **not** with the expected meaning. Actions have an effect on the system in which the policies are applied. Several operators are available to compose actions:

and: leads to the execution of both actions in either order;

andthen: specifies that the first action precedes the second in any execution;

or: specifies that either one of the actions should be taken;

orelse: is like **or** but prescribes that the first option is preferred.

In all these, an action may be atomic or composite.

Triggers and actions are domain specific atoms. Conditions are either domain specific or more generic (e.g. time) predicates. The following describes the syntax of the policy language:

```
polrule    ::= appliesto location [when triggers] [if conditions] do actions
```

```
triggers   ::= trigger | triggers or triggers
conditions ::= condition | not conditions |
               conditions or conditions | conditions and conditions
actions    ::= action | actions actionop actions
actionop   ::= and | or | andthen | orelse
```

3.4.2 Policy Attributes

The principal means to adapt a workflow to the needs of a stakeholder is by describing the behaviour of tasks using policies. To the StPowla user, these characteristics appear as *attributes*, i.e. properties of either tasks or workflows. Attributes can be introduced at different times: they can be *predefined*, i.e. they come with StPowla and are applicable to any task or workflow, or they can be *domain specific*, i.e. they are part of the ontology of a particular business domain.

The StPowla user can refer to the state of the execution of the workflow in terms of *attributes*, i.e. properties of individual tasks or of the whole workflow. Most of the attributes are part of the *domain specific* specialization of the Appel component of StPowla, i.e. they come from the ontology of a particular business domain. Finally, each task can have its own attributes.

Predefined attributes include:

automation which facilitates constraining how the task is to be executed;

actorRole which is bound to the role interacting with the system to fulfill a task, if any;

actorId which is bound to the identity of the actor playing the requested role, if any.

Attributes have types. For instance, automation takes values in {automatic, interactive}, and the first value excludes the involvement of humans in the fulfillment of the task. actorId is a String, as is actorRole, which has “user” as default value.

In general, attributes can be overridden by more specific definitions. For instance, the `actorRole` type may be redefined according to the specifics of the business, in a domain dependent definition section. Moreover, in two workflows they may be different, as defined in a workflow dependent definition section.

Some attributes related to a task may be already bound and available on task entry, as task parameters; some other may depend on the results of the invoked service[4]. They are used in policies attached to subsequent tasks.

Similarly, some workflow attributes are available at activation, as workflow parameters, while other are computed along execution. For instance, a workflow relating to a bank, may have an attribute `branchSize` ranging in {`small`, `medium`, `large`}, which is bound when the workflow is instantiated.

An attribute is accessed by a policy, with syntax

```
<prefix>.<attributeName>
```

where the prefix is either the name of a task, or is left empty, in which case the current task or the current workflow is assumed. In case of ambiguity, the current task may be referenced as **`thisTask`**, and the current workflow as as **`thisWF`**.

Finally, here we assume that the standard operators for a type are available as policy actions. For instance, the `totalCost` attribute of a workflow may have an operation `inc` that can be used to accumulate the costs of its tasks.

[4]In this case, the attributes will be eventually refined along the development process to become part of the type of the value that the invoked service returns. However, at the business level of abstraction, these are seen as related to the task, for their use in policies.

3.5 Tasks and Services

Tasks are the items where BPM and SOA converge, and where variability occurs by using policies: the intuitive notion of task is revisited to offer the novel combination of services and policies. To specify tasks, we specialize Appel to deal with services, by introducing a special action for service discovery and invocation.

A task has a *name*, which is intended to convey its purpose. For instance, “makeCoffee” is the name of a task where a coffee is expected to be prepared. The details of the working of the task are detailed as the understanding of the workflow grows. In well established domains, each task name should identify precise requirements, e.g. the broad functionality of “makeCoffee” can be easily inferred from the name. The task also has an associated policy in Appel that uses the name of the task to express the functional requirements in the service choice, and also specifies non-functional requirements.

A *default* policy is associated with each task. It says that when the control reaches the task, a service is looked for, bound and invoked, to perform the main functionality of the task. As such, a task must have a policy in order to execute, i.e. the policy actions the task. The syntax is as follows:

```
policy <taskName>.default is
appliesTo <taskName>
     when taskEntry(<args>)
       do req(main, <args>, [])
```

For instance, the default policy of a task (with no arguments) where a coffee has to be made is the following one:

```
policy makeCoffee.default is
appliesTo makeCoffee
     when taskEntry([])
```

```
do req(main, [], [])
```

With **taskEntry** we denote the policy trigger, whose arguments are the task parameters, if any. Action `req( _,_,_ )` is the essential bit of the Appel specialisation to deal with selection and invocation of services. It is generic, i.e. independent of the business domain. This action takes three arguments:

- the type of the service, expressing its basic functionality. By default it coincides with the name of the task, and is denoted simply as **main**. Anyway, the type must be known in the domain description;
- the list of service parameters, in terms of the task parameters and attributes;
- the specification of the constraints on service selection: they express Service Level Requirements[5]. In the default policy the list of constraints is empty: any service of the required type will do.

Definition 1 T*he semantics of the* req *action are: find a service as described by the first and third arguments, bind to it, and invoke it with the values in the second argument*[6]*. The* action succeeds *if a service is found, and its invocation is successful. It* fails *if either no service is found or if the bound service fails. The binding acts as a commit: only one service is invoked, and if its invocation fails no other found service is invoked.*

Adaptation occurs when the user overrides the default policy with his own, by specifying the SLA constraints, or by using the composition operators of Appel.

Let us show some examples. In case of task makeCoffee, a constraint on service invocation might deal with an attribute `CupTemperature`, which can take values in {`cold`, `warm`}. The request for coffee can be refined, to request that the coffee is served in a warm cup, by introducing the following policy.

[5] these are used as the requirements to precede a Service Level Agreement.

[6] we assume an automatic search and matching of services to tasks, thus allowing the user to work without the need for detailed service knowledge.

```
policy makeCoffee.default is
appliesTo makeCoffee
     when taskEntry([])
       do req(main, [], [CupTemperature = warm])
```

Service discovery must now take into account the SLA constraints, in the third argument of **req**: the invoked service must be able to satisfy the given constraint on CupTemperature.

SLA constraints usually address different kinds of concerns, or SLA *dimensions*. A dimension is specified in the domain description by giving it a name, the set of values and the applicable operators (if different from the generic ones, like equality and inequality, which we always assume). In the coffee example, one dimension is `CupTemperature`, with values in {`cold`, `warm`}. Essentially, a dimension defines a type.

Since Appel permits actions to be combined, we can provide the customer with a glass of water (at no cost) before the coffee:

```
policy makeCoffee.default is
appliesTo makeCoffee
     when taskEntry([])
       do   req(glassOfWater, [], [Cost = 0])
          andthen
            req(main, [], [])
```

The operator `andthen` lets the two services be invoked in sequence.

It is useful to define repairing actions, to cope with failures (e.g. no service found or service invocation failure). Operator `orelse` permits the introduction of repairing actions: if it is not possible to have a coffee then a tea is looked for.

```
policy makeCoffee.default is
```

```
appliesTo makeCoffee
     when taskEntry([])
       do   req(main, [], [])
          orelse
            req(makeTea, [], [])
```

In all the previous examples the policy has no conditional clause, i.e. it is always applicable. In practice, one may want to subordinate the invocation of a service, to some condition, such as:

```
policy makeCoffee.default is
appliesTo makeCoffee
     when taskEntry([mood])
       if mood=sleepy
       do req(main, [], [])
```

Here, the task is passed through, without doing anything, if the customer is not sleepy and hence does not need a coffee. So, the actual workflow can depend on the state of the system, inspected by exploiting attributes.

We can now provide the definition of task success and failure.

Definition 2 *A task* succeeds *if the associated policy is either not applicable, i.e. its conditions are not satisfied, or if its action succeeds. The task* fails *if the policy is applicable but its action fails.*

In the former example the task fails if we are sleepy, and the req action fails: no service is found in our SLA conditions or a service is found but its execution fails. Conversely, the task succeeds if a service is found and correctly executed, or if the policy is not applicable, since we are nicely awake.

In Appel policies can be combined in groups that control the order in which applicability is checked. Operator **seq** checks its second argument only if the first one is not applicable: caffeine comes in a coke only if the customer is thirsty.

```
policy makeCoffee.default is
appliesTo makeCoffee
     when taskEntry([mood])
       if mood != thirsty
       do   req(main, [], [])
          seq
            when taskEntry([])
              do req(glassOfCoke, [], [])
```

In StPowla, we exploit this feature to allow the user to assign priorities to the policies attached to the same task. We envisage an extension of the graphical interface to Appel, known as the Appel wizard [87], to support the user of StPowla in policy management, e.g. with respect to priorities among policies.

In the following example the Appel choice operator is used: if `morning` evaluates to `true`, the first of the two rules will be applied, otherwise the second one will be applied.

```
policy makeCoffee.default is
appliesTo makeCoffee
     when task_entry
       do   req(main, _ )
          g(morning)
            req(decaffeinated, _ )
```

3.6 The Service Level Agreement Language

The purpose of the third argument of `req` is to specify a *list* of constraints on service selection. We can constrain a dimension to a single value, or to a range of values. Service Level Agreement (SLA) constraints are expressed using parameters, attributes and *get* functions that permit the workflow management system to inspect the state of the workflow. Differently from the conditions in the if clause, which are evaluated when executing a policy, these conditions are evaluated by the `req` action itself during service selection (and ultimately by the service broker component), against each candidate service. Should the condition be independent from the current state, as in

```
Automation = automatic
```

we could simply consider the type of the third argument to be a String, and let `req` interpret it. However, one may want to express constraints that depend on the state of the computation, e.g. that the automation kind requested is defined by the current value of the corresponding attribute of the task. We need a mechanism to force partial evaluation of the condition in the third argument of `req`. The situation is similar to what happens with SQL queries in JDBC. We use the convention of prefixing a question mark to those part of a constraint that need evaluation. For instance, a constraint like:

```
Automation = ?(thisTask.automation)
```

will entail a search for an `automatic` or `interactive` service, according to the value of `thisTask.automation` when the policy is applied.

3.7 Pragmatics of the Customization

Our attitude is that, when a new policy is introduced, the user should define its relation with other policies applying to the same task. This can be done using the Appel operator `seq`, which introduces an enforcement. That is, we traverse the structure, determining whether the first policy is applicable: if so we apply it, otherwise we check the second one.

Looser attitudes, like composing policies in parallel, may lead to policy conflicts. Conflicts can be detected with a supporting tool [60], and their resolution can be manual or automatic. Manual resolution can include re-formulation of policies. A technique to automatically solve conflicts is to give a priority to the most recently introduced policies.

3.8 Summary

In this chapter, we have taken the three ingredients of Business Process Management (i.e. workflows), Policies and Service Oriented Architecture and combined them into a recipe we called StPowla.

Workflows define the core process in terms of tasks and control flows between tasks. The workflows themselves are described using a graphical notation that abstracts away any specific implementation or even technical detail, for the purpose of being appropriate at the business level, where most human users are indeed non-technical.

Policies describe variability of the core process by acting as a technology overlaid on workflows. SOA provides the underlying functionality. For this thesis, it has been assumed that the mapping of workflow tasks to SOA services is already provided, i.e. any task can be defined, with any requirement, and another independent system can find a Service (or composite Service) which provides the desired functionality.

In the next chapter, we will consider more closely policies and their (potential) impact upon workflows. More accurately, we will define specific effects that policies can have on workflows, which we shall identify as workflow reconfigurations.

Chapter 4

Workflow Reconfiguration

It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change. (Charles Darwin).

4.1 Introduction

In the previous chapter we presented StPowla as a combination of workflows, policies and Service Oriented Architecture (SOA) as a viable solution for defining variability separately to business processes. We progress from the initial description of StPowla to the closer examination of how policies interact with workflows. We have already noted that the combination of workflow tasks and services has been assumed and is thus out of scope for this thesis. In this chapter, we define a set of functions that can be specified by policies and performed over workflows in order to achieve some form of reconfiguration. We use informal descriptions, policy syntax and graph transformation rules to provide the set of definitions.

A *reconfiguration* is a functional change on a workflow, i.e. the actions of the workflow

change. The purpose of a reconfiguration is to allow the core process to react to changing contextual and environmental factors. For example, returning to the early example in Chapter 1 (Figure 1.2), there is a core process that is followed under most circumstances. Only when certain factors are present (i.e. a received order has a particularly high value), then a variation to the core process is carried out (Figure 1.4).

A reconfiguration is a *short-lived* modification to the workflow: the modification expires when the workflow instance ends and the same core process is used at the start of the next instance. This implies that the core process, defined by the workflow itself, is not permanently changed by policies. To achieve lasting change the workflow designer, i.e. the business analyst, must change the core model directly.

Previously we have seen examples of Event-Condition-Action (ECA) policies. These policies are rules that, triggered by a specific event and when satisfying some condition, perform specific actions. A task has a requirements list (recall the `req(main,[])` function) which we identify as the *implicit* actions to perform once it has started. The `req` function, although simply a function used as the action component of an ECA, is all part of the core StPowla workflow execution. We now extend the ECA and policy concepts to describing *explicit* functionality, i.e. additional to the core task functionality.

Qualitative changes can also be made to the workflow where constraints are placed over the execution of the workflow itself. Both functional and qualitative modification types are not mutually exclusive, i.e. a workflow can be reconfigured and qualitatively constrained simultaneously. Such is the breadth and depth of information we cover about reconfigurations that we do not consider qualitative changes in this thesis.

This chapter is structured as follows: firstly the concept of graph transformation rules is broadly defined in the context of our problem area. Secondly, a workflow ontology is described as a means to providing a location where contextual workflow information can be defined. Thirdly, each reconfiguration function is described using a combination of informal language, policy syntax and graph transformation rules.

4.2 Graph Transformation

It is commonly understood that (visual) modelling languages, as tools that enable easier development and specification of software systems, can be seen as graphs (e.g. [22, 95]). Thus when a model changes from one state to another, a graph transformation of some kind is involved.

A StPowla workflow, or indeed any process, can be seen as a network of nodes and directional connecting edges (i.e. a graph). Workflow items are tasks or operators, and these are each represented by a single node. Edges represent the control flows between these items, where one control flow is shown as one edge. Since graph transformations are typically applied to a meta-model, we can apply a similar principle to the process model.

Policies change the graph in some way and therefore graph transformation becomes a logical tool for the specification of reconfigurations. Graph transformation supports a visual representation of rules, which is reminiscent of the intuitive way in which engineers would sketch, for example, network reconfigurations [47]. Considering also the likelihood of a wizard to help users define policies, we find a close match between graph transformation rules and policies at an abstract level.

We use graph transformations to visually describe the functions available to policies and thus the effect that policies can have on a workflow instance. A graph transformation rule takes a basic formalism (i.e. a typed graph), in our case a workflow model and its instance graph, in our case the executing workflow instance.

The graph transformation rule is defined in terms of the original typed graph (the *source*) and the resulting typed graph (the *target*) after the transformation has occurred. In addition, a similar step is applied for the instance graph.

A rule p is written formally as $p : L \to R$ with $L \cap R$ well defined in different presentations.

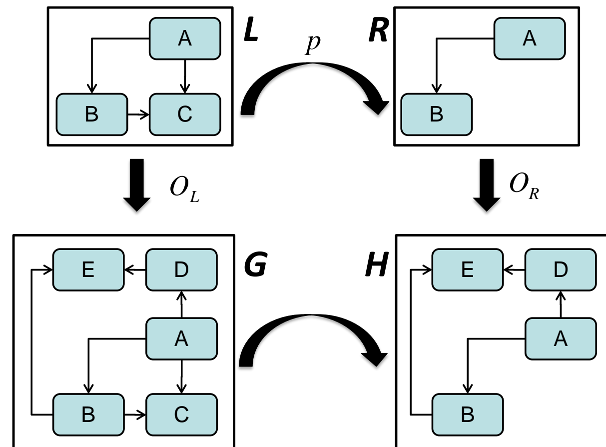


Figure 4.1: An example graph transformation rule.

In StPowla language, L represents a part[1] of the executing workflow instance and R represents what it will transform into, according to the rule p. An example visualization of graph transformation rules is given in Figure 4.1. The rule simply removes node C and its relations to nodes A and B (as shown in L and R) in graph G in order to achieve graph H.

The subject part of the workflow model L is shown in the top left, with the result of the transformation R in the top right. Furthermore, two additional diagrams G and H show the executing workflow instance in its initial state (G) and then its fully transformed state (H). Thus there are mappings between L and G on the source side and between R and H on the transformed side.

The steps to performing a graph transformation are straightforward:

1. Select rule $p : L \rightarrow R$; occurrence $o_L : L \rightarrow G$;
2. Remove from G the occurrence of $L \backslash R$;
3. Add to result a copy of $R \backslash L$.

The purpose of defining policies in terms of graph transformations is threefold:

[1]It is possible for the source graph to display the entirety of the workflow but this would be counter productive due to the size of the graph that would often be required and also the lack of support for graph transformations over recurring patterns inside the workflow

First, graph transformation is known to be a scalable means of modifying graphs according to given rules. In this situation, the size of workflows and the quantity and complexity of policies can increase significantly, therefore requiring a scalable transformation mechanism.

Second, graph transformation visualizes the effects of policy functions. It also provides a means to visually and intuitively demonstrating the effects of combining multiple rules and is therefore useful for analysis purposes.

Third, the visual perspective of graph transformation rules is in keeping with the graphical design of the workflow. A policy creation wizard can incorporate a graph transformation viewer in order to ease the design of policies at an abstract level.

4.3 Workflow Ontology

Policy authors require some basic knowledge of the subject workflows in order to write policies that are effective. For example, one cannot write a policy for a specific task without knowing the task name and one cannot address a specific task attribute without knowing the attribute's name. Therefore, we provide a simple ontology for a workflow that contains definitions for the information that is exposed to policy authors and, by implication, to policies.

```
Workflow <name> is
  Types
    <name>: { type construction }
    ...
  Invariants
    <type> <invariant_name> = value
    ...
  Actors/Entities
    <name>
      Attributes
        <type> <attribute_name> [:= default_value]
        ...
    ...
```

```
Global Attributes
  <type> <attribute_name> [:= default_value]
  ...
Domain Dependent Attributes
  <type> <attribute_name> [:= default_value]
  ...
Workflow Attributes
  <type> <attribute_name> [:= default_value]
  ...
Tasks
  <name>
    Attributes
      <type> <attribute_name> [:= default_value]
      ...
  ...
Scopes
  <name> : [owner_name.]<name_start_item>,
             [owner_name.]<name_end_item>
  ...
```

According to this ontology, a workflow can be described according to a set of invariants, types, properties (global, domain dependent and workflow) and constructs (tasks and scopes). We note that the list of tasks does not include information on how tasks connect to each other, i.e. no control flow is defined inside the ontology - this is left for the visual workflow model.

Types are compositions of simple types (strings, dates, lists and numbers). At this conceptual stage, there is no need to expand upon this further to include additional types, e.g. collections, but it is enough for us to say that further types are possible. The syntax of defining a new type is as follows:

```
myNewType :
{
    <type> <attributeName_1> [:= default_value]
    ...
    <type> <attributeName_n> [:= default_value]
}
```

Type attributes can be accessed using common syntax `<type_instance_name>.<type_attribute_name>`.

Actors and entities are different stakeholders in the workflow, e.g. the supplier workflow includes the supplier and customer as two actors. Each of these may have attributes with optional default values, accessed in a similar way to types.

Global attributes are those that are accessible throughout the workflow system, including all workflows and their tasks. Domain dependent attributes are a specific category that can be used in accordance with APPEL, which separates the core policy language from the domain dependent implementation. Workflow attributes are specific to this particular workflow and also provide a separation from global attributes. Tasks are listed by name, together with their exposed attributes.

The main advantage of swimlanes is to provide a more global view of the executing workflow. Since workflows are typically focussed on the activities of only one actor/entity, it can be useful to see that workflow in light of other workflows from other actors/entities that it interacts with. Thus a swimlane is a graphical boundary that separates the specific workflow of one actor/entity from another.

Finally, composite tasks are listed under *Scopes*[2], for which each definition includes a start point, an end point, and all items in between the two. Since a number of workflows include the concept of swimlanes (graphic boundaries that identify each task inside with a specific actor or entity), we also allow swimlanes and optionally use the notation `OwnerName.TaskName` to uniquely identify a task according to the swimlane it is in. Thus, a swimlane is usually representative of an entity.

4.4 Reconfiguration Functions

Reconfigurations to the workflow are made through the result of calling functions over the workflow instance. These functions are called in policies according to parameters defined by the policy author. In this section, we define closely each reconfiguration function that is available and all necessary variations of those functions.

[2]Previously we said that a series of consecutive tasks, together with their interlinking control flows and operators could be considered as a composite task. The same principle can be identified as a scope, although a scope is typically not redrawn in a condensed way in the workflow. Instead, the scope is an outline of an area of the workflow, including all items inside that area.

Logically, a function is defined as $Func :: P \xrightarrow{params} P$, where P is a process (recalling the previous grammar). The list of parameters *params* may be passed to configure the transformation that the function is trying to achieve.

Broadly, we define the set of reconfiguration functions in Table 4.1.

Type	*Description*
Insert:	inserts a set of items into the workflow instance;
Delete:	removes an existing task from the workflow instance;
Fail:	designates an executing workflow task instance as having failed;
Abort:	designates an executing workflow task instance as having been aborted;
Block:	delays a task's execution.

Table 4.1: Reconfiguration functions

These functions enable end users to make almost unlimited changes to the entire workflow. The two examples in Chapter 1 described both an empty workflow that was built according to need and another that was comprehensively described. The former example is now possible through StPowla. We now define each function with their respective policy syntax, the logical process grammar and its graph transformation rule.

For clarity, the term *process* is used to describe a scope and the term *item* is used to describe any element that can be inside a workflow control flow, i.e. a task, composite task or operator. Graph transformation rules are given only with graphs L and R for brevity. The application of a set of these rules is illustrated in Chapter 6.

Both graphs L and R may contain "...", signifying that preceding and proceeding workflow components may exist and we are not concerned with what they actually are. Functions involving composite tasks are included to represent situations that require special attention compared to simple tasks[3]. Also, process diagrams may include tasks in L which are not connected to any other part of the workflow. These represent tasks that are available in a repository, but are not being used by the workflow.

[3]For example, any composite task can be regarded as a composition of a Flow Split operator followed by a Conditional Merge operator. In this situation, we must take into account both these operators, plus the items contained inside the connecting control flow, which could ultimately contain more operators and tasks.

4.4.1 Insert

The function *insert* places a pre-defined process in the executing workflow instance at a specific position. The function also allows the specification of whether the new process is to be inserted in parallel with or following the position. The position is referenced by the name of a target workflow item.

Insert is certainly the most complex of all the reconfiguration functions since it has so many variations. A user can insert a simple task in sequence with another or in parallel. The other task may be simple or composite. A user can also insert a process, which can include a number of operators, in sequence with a task or in parallel to it. Again, that existing task may be simple or composite.

Inserting a Task

We consider first inserting a simple task. To use the function in a policy, the following syntax should be used: *insert(Process* p_1*, Process* p_2*, boolean inParallel)*. For example:

```
policy EnsureGoodDrinkingTemperature is
    appliesTo MakeCoffeeWorkflow
         when PourCoffee.task_ended
           if coffee.temperature > 60
           do insert(AddDashColdWater, PourCoffee, false)
```

The relevant workflow ontology is as follows:

```
Workflow MakeCoffeeWorkflow is
    Actors/Entities
        coffee
            Attributes
                integer temperature := 0
    Tasks
        PourCoffee
        AddDashColdWater
```

This policy affects the `MakeCoffeeWorkflow`, an entire workflow dedicated to the steps of making a cup of coffee. After pouring coffee into a mug (the *trigger*), if

the temperature of the coffee (the *condition*) is greater than 60 °C[4], then a new task (`AddDashColdWater`) to add a dash of cold water (the *action*) is inserted into the executing workflow after the `PourCoffee` task[5]. This is done after the task has been completed but before the next task is started.

When inserting in sequence to an existing task like in this example, we are required to break the control flow prior to the existing task, map that to the incoming point of the new task, and map the outgoing control flow of the new task to the task or item that originally followed the existing task.

When inserting in parallel to an existing task, we are also required to insert a Flow Split operator before the existing task, in order to provide the parallel control flow path, plus a Conditional Merge operator following the existing task, to ensure that the control flow integrity is not compromised. This Conditional Merge is defined such that it receives two incoming mandatory control flows and shall only allow execution to proceed when both have completed (i.e. the number is 2)[6].

The policy author though should not be concerned with these aspects so the insertion of the appropriate flow split and conditional join operators is invisible to them, i.e. it is automatically done by the policy engine.

One might note that although it would not be impossible to insert a new task in parallel to one that has already finished executing, it would be a redundant change. However, since the functions are based on patterns, it may be possible that a second instance of the same task exists later on in the workflow and the policy author wishes to affect primarily the second instance.

Already we can see a number of variations to the insert function. The full list of variations is described in Table 4.2. This includes identifying if the new process is an atomic (simple)

[4] Since the attribute's type is an integer, we assume the value is a measure of degrees Celsius.
[5] Items coffee and temperature are defined in the workflow ontology.
[6] We note there is a redundancy here, but for the purpose of clarity, this has been ignored

or composite task, the existing task is an atomic or composite task, and if the new task is to be inserted in sequence or in parallel.

	Process p_1	*Process p_2*	*In Parallel*	*ID*
Insert	Atomic Task	Atomic Task	True	I_1
			False	I_2
		Composite Task	True	I_3
			False	I_4
	Composite Task	Atomic Task	True	I_5
			False	I_6
		Composite Task	True	I_7
			False	I_8

Table 4.2: Variations of the Insert function.

We continue to examine each of these variations individually for completeness.

Insert Variation I_1: This variation inserts a new atomic task to be placed in parallel with an existing atomic task. As previously discussed, this must also implicitly, i.e. away from the responsibility of the policy author, insert a Flow Split operator and a Conditional Merge operator. The policy syntax is *insert(T_2, T_1, true)*[7] and the grammar used to express this transformation is *Insert*:: $T_1 \xrightarrow{(T_2,true)} FJ(2,\{T_1,true\},\{T_2,true\})$.

This is defined visually as a graph transformation rule as in Figure 4.2.

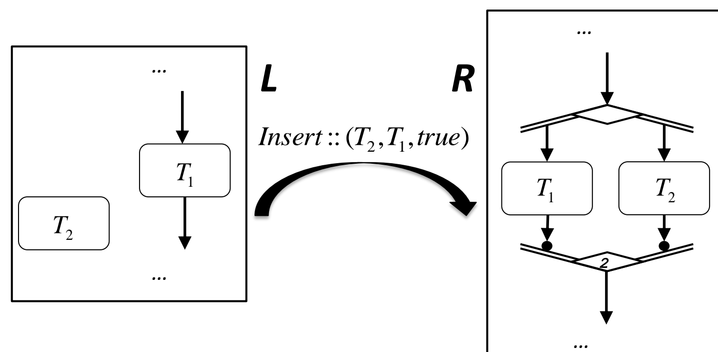


Figure 4.2: GT rule for insert variation I_1.

[7]the literal translation is "insert task T_2 in parallel with task T_1"

To illustrate this reconfiguration, we return to the example given in Chapter 1 and repeated in Figure 4.3, but without the customer's swimlane.

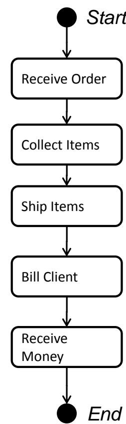


Figure 4.3: The full workflow for the supplier.

Now we add the following policy:

```
policy CheckRequestDeposit is
    appliesTo ReceiveOrder
         when task_completed
           if order.value > 1,000,000
           do insert(RequestDeposit, ReceiveOrder, false)
```

We also consider the related ontology:

```
Workflow SupplierWorkflow is
    Actors/Entities
        order
            Attributes
                integer value
    Tasks
        ReceiveOrder
        RequestDeposit
        ... (other tasks left for brevity)
```

This policy can be expressed as a graph transformation rule, with L, R, G and H graphs as shown in Figure 4.4. Importantly, graph L shows the source workflow items and what they should be transformed to in R. Graph G shows the full source workflow and H shows the result of applying the graph transformation rule to the original.

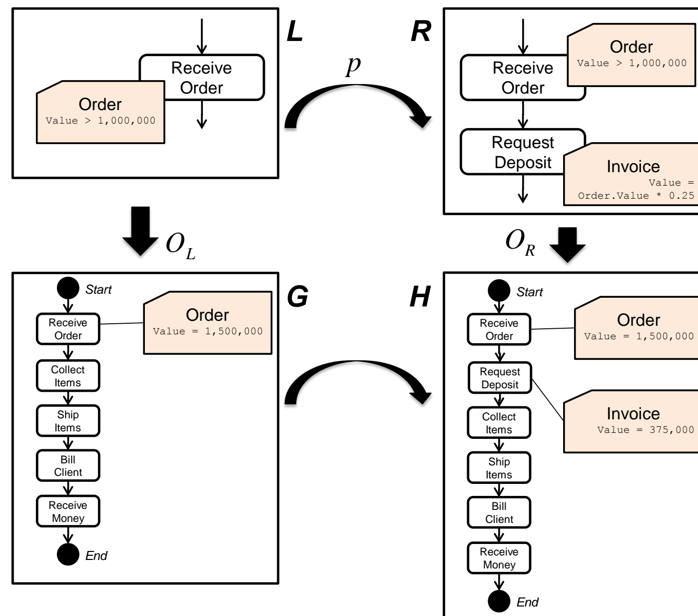


Figure 4.4: The graph transformation rule for the supplier workflow.

Insert Variation I_2: This variation inserts a new instance of an atomic task to be placed after and in sequence with an existing atomic task. The new task's incoming control flow is received from the referenced existing task's output. The new task's outgoing control flow is mapped to the next task in the workflow.

The policy syntax is *insert(T_2, T_1, false)*[8] and the grammar used to express this transformation is *Insert*:: $T_1 \xrightarrow{(T_2,false)} T_1; T_2$.

This is defined visually as a graph transformation rule as in Figure 4.5.

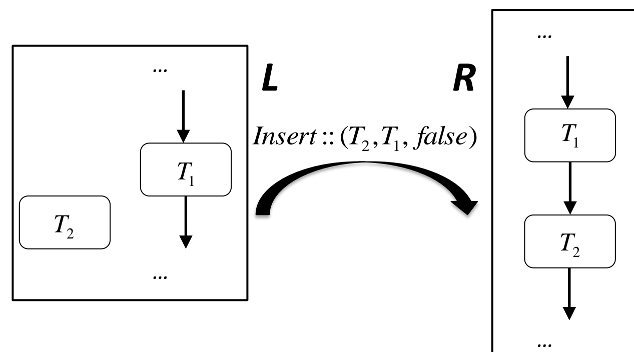


Figure 4.5: GT rule for insert variation I_2.

[8]the literal translation is "insert task T_2 after task T_1"

Insert Variation I_3: This variation inserts a new atomic task in parallel with an existing composite task (the minor difference to variation I_1 being the composition). Similarly, the policy syntax is *insert(T, CT, true)*[9] and the grammar used to express this transformation is $Insert:: T \xrightarrow{(CT,true)} FJ(2,\{CT,true\},\{T,true\})$.

This is defined visually as a graph transformation rule as in Figure 4.6.

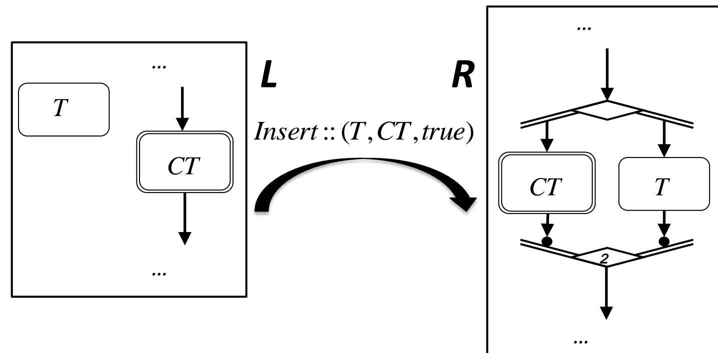


Figure 4.6: GT rule for insert variation I_3.

Insert Variation I_4: This variation inserts a new atomic task immediately after an existing composite task (similar to variation I_2 with the minor difference of composition). The policy syntax is *insert(T, CT, false)*[10] and the grammar used to express this transformation is $Insert:: T \xrightarrow{(CT,false)} T;CT$.

This is defined visually as a graph transformation rule as in Figure 4.7.

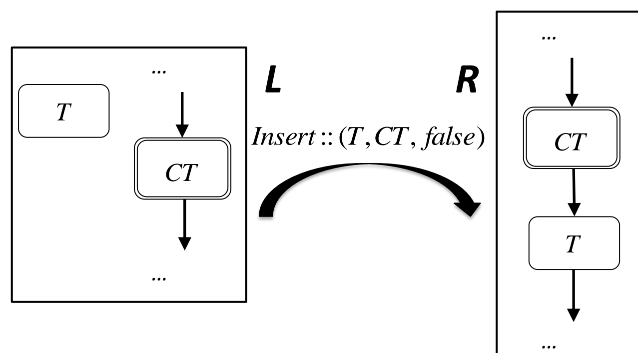


Figure 4.7: GT rule for insert variation I_4.

[9]the literal translation is "insert task T in parallel with composite task CT"
[10]the literal translation is "insert task T in sequence with composite task CT"

Remaining Insert Variations: The remaining variations follow similarly from the presented variations, with the difference being that the task to be inserted is a composite task. Each of the variations I_5, I_6, I_7 and I_8 are described here for completeness.

Insert Variation I_5 :

Policy syntax: *insert(CT, T, true)*

Literal translation: "insert CT to execute in parallel with T"

Grammar: *Insert*:: $T \xrightarrow{(CT,true)} FJ(2,\{T,true\},\{CT,true\})$

Graph Transformation Rule: See Figure 4.8.

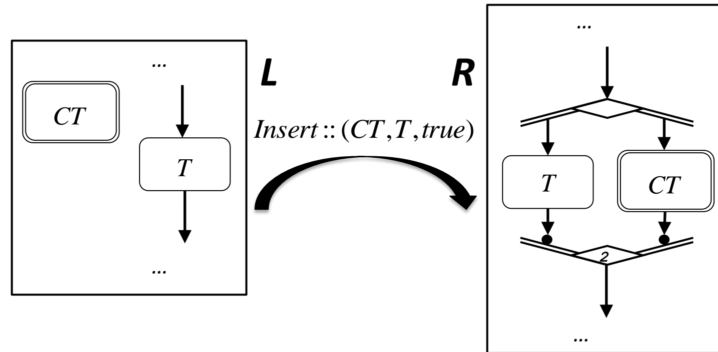


Figure 4.8: GT rule for insert variation I_5.

Insert Variation I_6 :

Policy syntax: *insert(CT, T, false)*

Literal translation: "insert CT to execute immediately after T"

Grammar: *Insert*:: $T \xrightarrow{(CT,false)} T;CT$

Graph Transformation Rule: See Figure 4.9.

Insert Variation I_7 :

Policy syntax: *insert(CT_2, CT_1, true)*

Literal translation: "insert CT_2 to execute in parallel with CT_1"

Grammar: *Insert* :: $CT_1 \xrightarrow{(CT_2,true)} FJ(2,\{CT_1,true\},\{CT_2,true\})$

Graph Transformation Rule: See Figure 4.10.

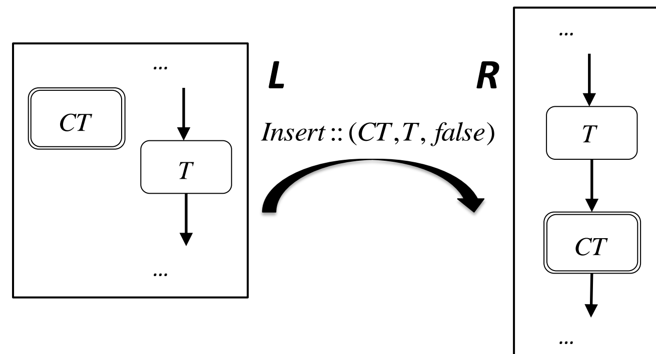


Figure 4.9: GT rule for insert variation I_6.

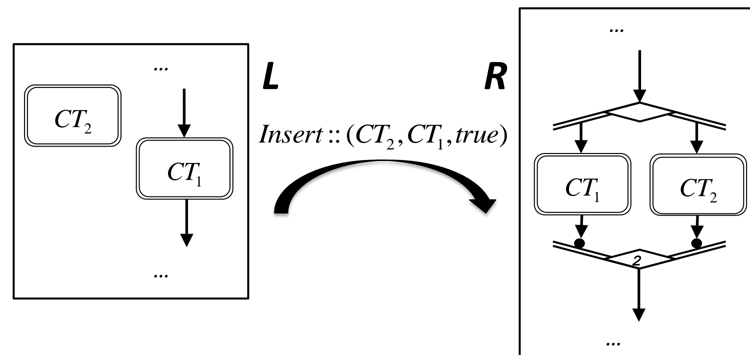


Figure 4.10: RGT rule for insert variation I_7.

Insert Variation I_8 :

Policy syntax: *insert(*CT_2*,* CT_1*, false)*

Literal translation: "insert CT_2 to execute immediately after CT_1"

Grammar: *Insert*:: $CT_1 \xrightarrow{(CT_2, false)} CT_1; CT_2$

Graph Transformation Rule: See Figure 4.11.

4.4.2 Inserting Operators

So far in this chapter, we have discussed and demonstrated how to insert atomic and composite tasks into the workflow at any given point, whether the new process is to execute in parallel to an existing reference point or in sequence to it. We now process to give consideration to the insertion of workflow operators, namely Flow Junction, Flow Split, Strict Preference and Random Choice.

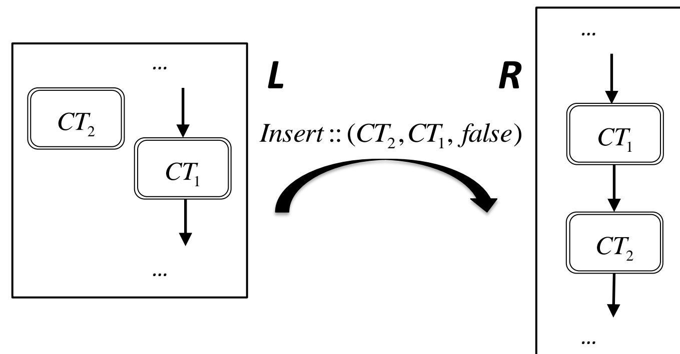


Figure 4.11: RGT rule for insert variation I_8.

The insertion of a workflow operator is not trivial; the integrity of the workflow must be maintained at all times and therefore if a new operator enforces a structural change in the workflow, it must ensure that the workflow can resume to the final end point somehow. We must also consider that the insertion of an operator can only have the desired change over the pattern of the workflow that is intended by the policy author (we assume that the author is perfect in their policy design and definition).

Inserting Flow Junction

The Flow Junction operator directs the workflow control flow down one of two available paths, with the choice being dependent upon the outcome of a boolean test. Recalling the grammar, Flow Junction is $\lambda^{?}P : P$, but it also implicitly includes a simple (XOR-type) join, enabling the operator(s) to be inserted where there is only one control flow. The implication here is that although there are two processes included in the Flow Junction operator, they must both converge to one single point as the workflow continues passed this operator.

We identify two variations to this rule: 1) insertion of the operator in sequence with an existing workflow item; or 2) insertion of the operator in parallel with an existing workflow item. Furthermore, we can consider both atomic tasks or composite tasks being referenced as the workflow item.

For completeness, we provide all variations for this rule according to Table 4.3, each given with a variation Id. For each variation, we provide a description, policy syntax and grammar (literal translations omitted).

For ease of use, it can be assumed that when a flow junction is inserted, both new control flows include a single *empty* task with names of the form *<flow-junction-operator-name_dummy1>* and *<flow-junction-operator-name_dummy2>*, respectively. We also make the following assumptions:

- The name of the new Flow Junction operator is FJ_{new};
- The name of the new Flow Merge operator is SJ_{new};
- The name of the existing atomic task is T;
- The name of the existing composite task is CT.

	With Respect To	*In Parallel*	*Id*
Insert Flow Junction	Task	True	IFJ_1
		False	IFJ_2
	Composite Task	True	IFJ_3
		False	IFJ_4

Table 4.3: Insert Flow Junction Variations.

Insert Flow Junction Variation IFJ_1 :

Description: Inserts a Flow Junction and Flow Merge in parallel with an existing atomic task.

Policy syntax: *insert(FJ_{new}, T, true)*

Grammar: *Insert:: T* $\xrightarrow{((\lambda^? FJ_{new}_dummy1 : FJ_{new}_dummy2), true)}$

$$FJ(2, \{T, true\}, \{(\lambda^? FJ_{new}_dummy1 : FJ_{new}_dummy2)\}, true)$$

Graph Transformation: See Figure 4.12.

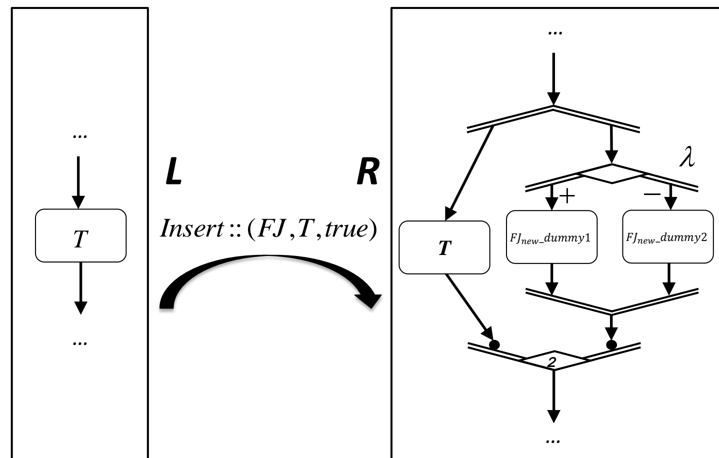


Figure 4.12: GT rule for insert Flow Junction variation IFJ_1.

Insert Flow Junction Variation IFJ_2 :

Description: Inserts a Flow Junction and Flow Merge in sequence with an existing atomic task.

Policy syntax: *insert(FJ_{new}, T, false)*

Grammar: *Insert::* $T \xrightarrow{((\lambda^{?} FJ_{new}_dummy1 : FJ_{new}_dummy2), false)}$

$$T; (\lambda^{?} FJ_{new}_dummy1 : FJ_{new}_dummy2)$$

Graph Transformation: See Figure 4.13.

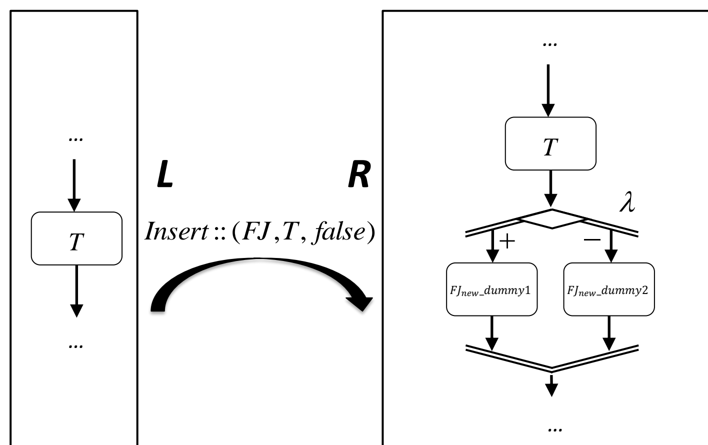


Figure 4.13: GT rule for insert Flow Junction variation IFJ_2.

Insert Flow Junction Variation IFJ_3 :

Description: Inserts a Flow Junction and Flow Merge in parallel with an existing

composite task.

Policy syntax: *insert(FJ_{new}, CT, true)*

Grammar: *Insert:: CT* $\xrightarrow{((\lambda^{?}FJ_{new}_dummy1:FJ_{new}_dummy2),true)}$

$$FJ(2,\{CT,true\},\{\lambda^{?}FJ_{new}_dummy1 : FJ_{new}_dummy2\},true)$$

Graph Transformation: See Figure 4.14.

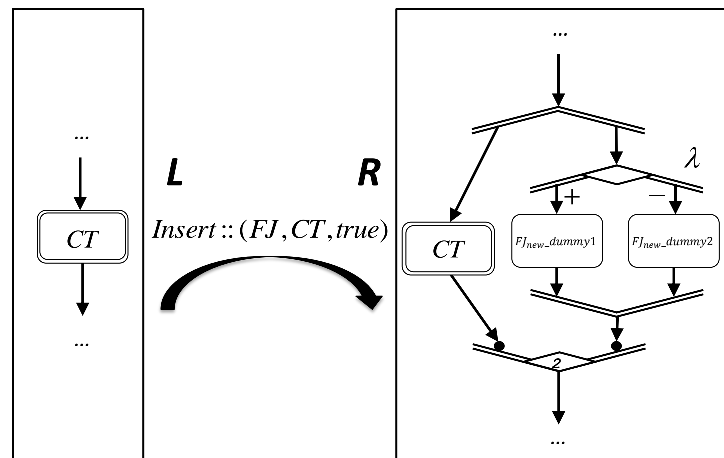


Figure 4.14: GT rule for insert Flow Junction variation IFJ_3.

Insert Flow Junction Variation IFJ_4 :

Description: Inserts a Flow Junction and Flow Merge in sequence with an existing composite task.

Policy syntax: *insert(FJ_{new}, CT, false)*

Grammar: *Insert:: CT* $\xrightarrow{((\lambda^{?}FJ_{new}_dummy1:FJ_{new}_dummy2),false)}$

$$CT;(\lambda^{?}FJ_{new}_dummy1 : FJ_{new}_dummy2)$$

Graph Transformation: See Figure 4.15.

Inserting Flow Split

The Flow Split operator allows the definition of several concurrent execution paths within the workflow. Since multiple paths are created from one, they must also converge into one

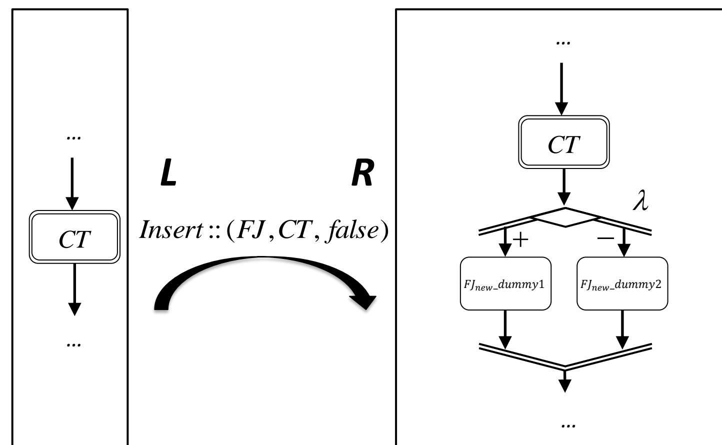


Figure 4.15: GT rule for insert Flow Junction variation IFJ_4.

path following completion of the operator. Thus, the Flow Split is always accompanied by a Conditional Merge. This merge operator will use default settings (where each incoming branch is mandatory and the number of required branches is equal to the number of incoming branches).

The purpose of this function is to allow multiple concurrent process flows to be inserted into the workflow at a given point. This may be in parallel with another set of existing process flows (i.e. in parallel with one path between an existing flow split and corresponding join operator), or in parallel to a newly-inserted process.

We place a strict constraint upon the insertion of a Flow Split operator: the operator shall only have two outgoing control flow paths. The reason for this constraint is that gaining further parallel paths is possible by re-using the function.

One such example can be considered in the supplier example. If the incoming order has a significant value, then the request for a deposit can be in parallel with collecting the items. At the same time, the supplier may need to restock the items and order in replacements. Alternatively, as a person is boiling the kettle in order to make coffee, they may also prepare breakfast by toasting bread and making scrambled eggs in parallel.

Note also that this function provides a different outcome to inserting a workflow task (atomic or composite) in parallel with an existing item. Although in the latter case paral-

lelism is created, this function allows parallelism to be created independently of current workflow items. However, one might also consider that this operator is a shorthand way of inserting a new, empty task in parallel or in sequence to an existing task, and then inserting a second empty task in parallel with the other empty task. The significant difference is that with this function, the policy author may define specific settings for the Conditional Merge operator whereas in the other scenario they cannot.

As with the Flow Junction operator, there are four variations of this rule depending on parallelism and the target related workflow entity. Those variations are shown in Table 4.4. Again, we provide a description, the policy syntax, the grammar and the graph transformation rule.

	With Respect To	*In Parallel*	*Id*
Insert Flow Split	Task	True	IFS_1
		False	IFS_2
	Composite Task	True	IFS_3
		False	IFS_4

Table 4.4: Insert Flow Split Variations.

Insert Flow Split Variation IFS_1 :

Description: Inserts a Flow Split and Conditional Merge in parallel with an existing atomic task.

Policy syntax: *insert(*FS_{new}*, T, true)*

Grammar: $Insert :: T \xrightarrow{(FS_{new}(2,\{FJ_{new}_dummy1,true\}:\{FJ_{new}_dummy2,true\}),true)}$

$$FJ(2,\{T,true\},\{F_{new}(2,\{FJ_{new}_dummy1,true\}:\{FJ_{new}_dummy2,true\}),true\})$$

Graph Transformation: See Figure 4.16.

Insert Flow Split Variation IFS_2 :

Description: Inserts a Flow Split and Conditional Merge in sequence with an existing atomic task.

Policy syntax: *insert(*FS_{new}*, T, false)*

Grammar: $Insert :: T \xrightarrow{(FS_{new}(2,\{FJ_{new}_dummy1,true\}:\{FJ_{new}_dummy2,true\}),false)}$

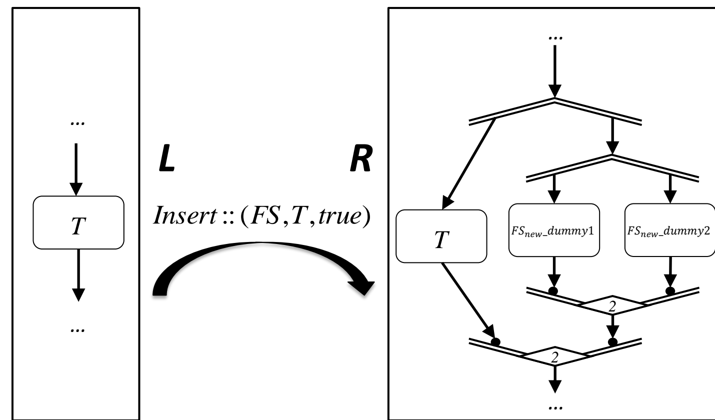


Figure 4.16: GT rule for insert Flow Split variation IFS_1.

$$T; FJ_{new}(2, \{FJ_{new}_dummy1, true\} : \{FJ_{new}_dummy2, true\})$$

Graph Transformation: See Figure 4.17.

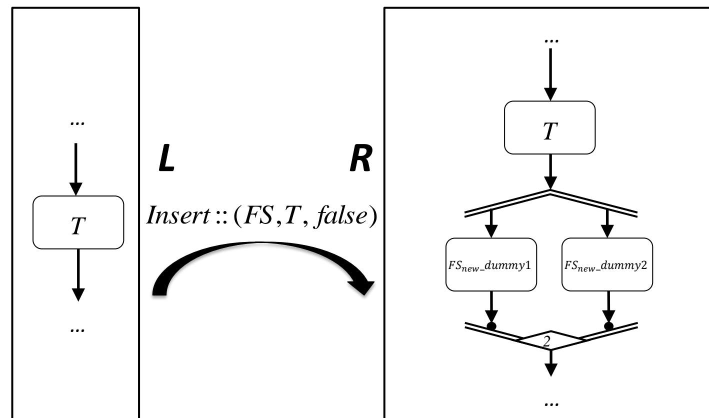


Figure 4.17: GT rule for insert Flow Split variation IFS_2.

Insert Flow Split Variation IFS_3 :

Description: Inserts a Flow Split and Conditional Merge in parallel with an existing composite task.

Policy syntax: *insert(FS_{new}, CT, true)*

Grammar: $Insert :: CT \xrightarrow{(FS_{new}(2, \{FJ_{new}_dummy1, true\} : \{FJ_{new}_dummy2, true\})), true}$

$$FJ(m, \{CT, true\}, \{FJ_{new}(2, FJ_{new}_dummy1 : FJ_{new}_dummy2), true\})$$

Graph Transformation: See Figure 4.18.

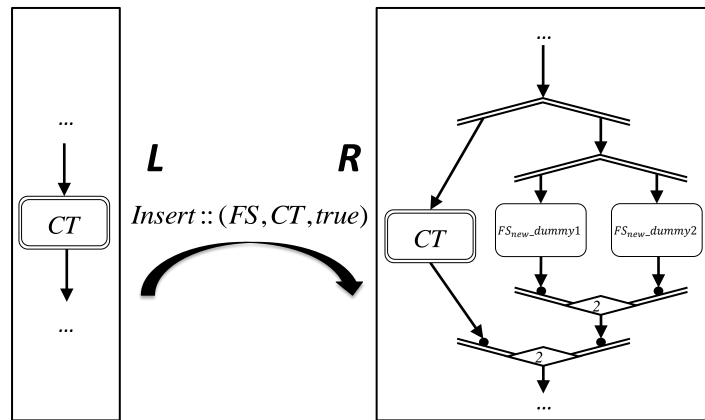


Figure 4.18: GT rule for insert Flow Split variation IFS_3.

Insert Flow Split Variation IFS_4 :

Description: Inserts a Flow Split and Conditional Merge in sequence with an existing composite task.

Policy syntax: *insert(FS_{new}, CT, false)*

Grammar: $Insert :: CT \xrightarrow{(FS_{new}(2,\{FJ_{new}_dummy1,true\}:\{FJ_{new}_dummy2),true\},false)}$

$$CT;(FJ_{new}(2,\{FJ_{new}_dummy1,true\}:\{FJ_{new}_dummy2,true\}))$$

Graph Transformation: See Figure 4.19.

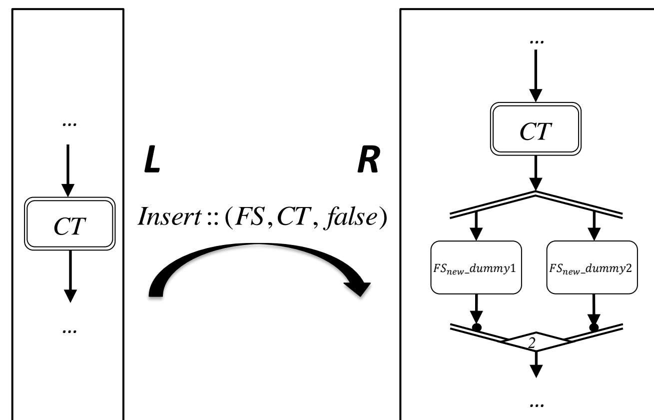


Figure 4.19: GT rule for insert Flow Split variation IFS_4.

Inserting Strict Preference

This operator specifies a set of atomic tasks to be executed sequentially according to a pre-defined order. The operator completes once one of those tasks completes, i.e. only after a task fails does the next task start. There is one outgoing flow per task, but it is possible to merge these into one flow as necessary. Since only one of the output flows is active, a Flow Merge is required to merge all of the flows into a single flow.

Following from the Flow Junction operator, there are four variations for the insertion of a Strict Preference operator inside the workflow (Table 4.5). We assume that the operator is predefined, although that does not restrict from the possibility of reconfiguring the output flows. For example, each of the output flows may be empty and only through additional policies can new tasks be inserted into them. For simplicity, we also assume that the operator has no error output flow. However, if there were, it would contain a dummy task and the flow would be incorporated to the final Flow Merge operator.

	With Respect To	*In Parallel*	*Id*
Insert Strict Preference	Task	True	ISP_1
		False	ISP_2
	Composite Task	True	ISP_3
		False	ISP_4

Table 4.5: Insert Strict Preference Variations.

We proceed to describe each of these variations in the same manner as the previous operators, using a description, the policy usage, the grammar and the graph transformation rule.

Insert Strict Preference Variation ISP_1 :

Description: Inserts a Strict Preference and Flow Merge in parallel with an existing atomic task.

Policy syntax: *insert(SP_{new}, T, true)*

Grammar: *Insert:: T* $\xrightarrow{(SP(\{spt_1,P\},\{spt_2,P\}),true)}$

$$FJ(2,\{T,true\},\{SP(\{spt_1,P\},\{spt_2,P\}),true\})$$

Graph Transformation: See Figure 4.20.

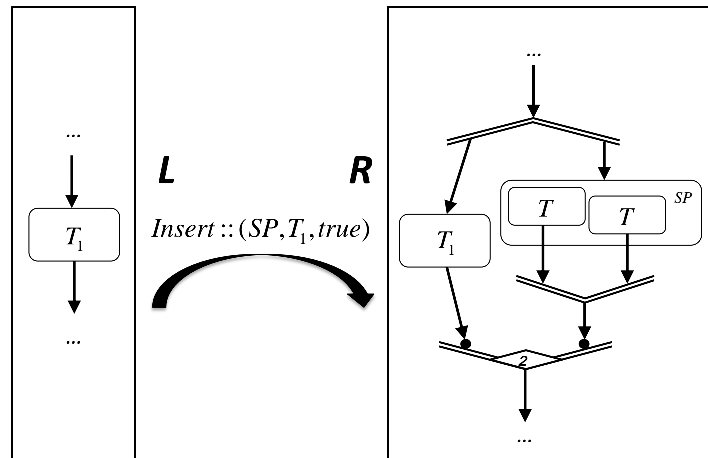


Figure 4.20: GT rule for insert Strict Preference variation ISP_1.

Insert Strict Preference Variation ISP_2 :

Description: Inserts a Strict Preference and Flow Merge in sequence with an existing atomic task.

Policy syntax: *insert(SP_{new}, T, false)*

Grammar: *Insert:: T* $\xrightarrow{(SP(\{spt_1,P\},\{spt_2,P\}),false)} T; SP(\{spt_1, P\}, \{spt_2, P\})$

Graph Transformation: See Figure 4.21.

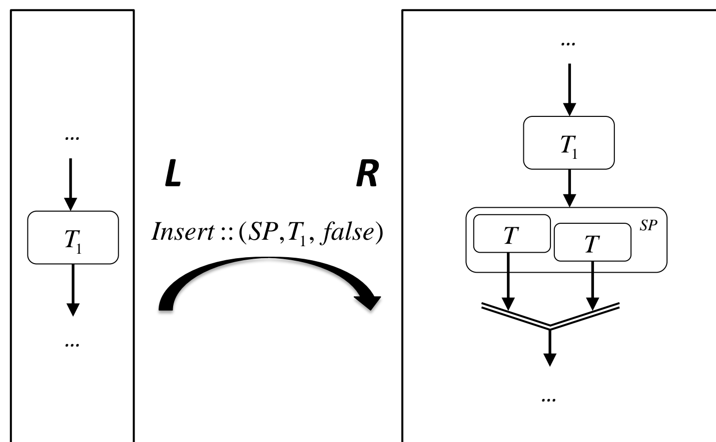


Figure 4.21: GT rule for insert Strict Preference variation ISP_2.

Insert Strict Preference Variation ISP_3 :

Description: Inserts a Strict Preference and Flow Merge in parallel with an existing composite task.

Policy syntax: *insert(SP$_{new}$, CT, true)*

Grammar: *Insert:: CT* $\xrightarrow{(SP(\{spt_1,P\},\{spt_2,P\}),true)}$

$$FJ(2, \{CT, true\}, \{SP(\{spt_1, P\}, \{spt_2, P\}), true\})$$

Graph Transformation: See Figure 4.22.

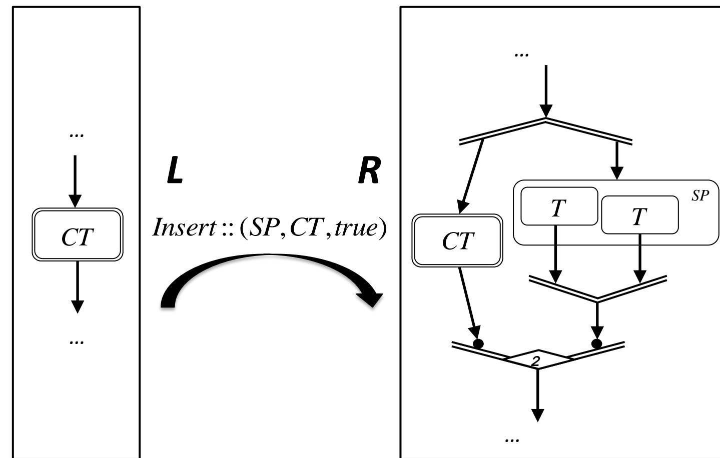


Figure 4.22: GT rule for insert Strict Preference variation ISP_3.

Insert Strict Preference Variation ISP_4 :

Description: Inserts a Strict Preference and Flow Merge in sequence with an existing composite task.

Policy syntax: *insert(SP$_{new}$, CT, false)*

Grammar: *Insert:: CT* $\xrightarrow{(SP(\{spt_1,P\},\{spt_2,P\}),false)} CT; SP(\{spt_1, P\}, \{spt_2, P\})$

Graph Transformation: See Figure 4.23.

Inserting Random Choice

This follows function works similarly to inserting the Strict Preference operator, even with an additional Flow Merge at the end. The specific variations for inserting a Random

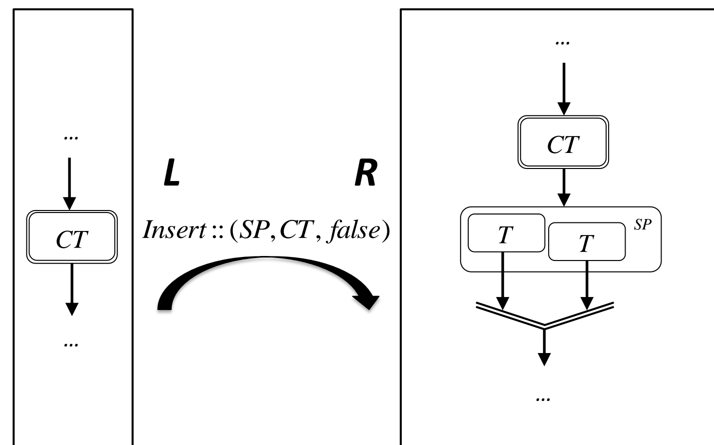


Figure 4.23: GT rule for insert Strict Preference variation ISP_4.

Choice operator in parallel with an existing task are shown in Table 4.6. Once more, for each variation we provide a description, the policy syntax, the grammar and a the graph transformation rule.

	With Respect To	*In Parallel*	*Id*
Insert Random Choice	Task	True	IRC_1
		False	IRC_2
	Composite Task	True	IRC_3
		False	IRC_4

Table 4.6: Insert Random Choice Variations.

Insert Random Choice Variation IRC_1 :

Description: Inserts a Random Choice and Flow Merge in parallel with an existing atomic task.

Policy syntax: *insert(RC_{new}, T, true)*

Grammar: *Insert:: T* $\xrightarrow{(RC(\{rct_1,P\},\{rct_2,P\}),true)}$

$$FJ(2,\{T,true\},\{RC(\{rct_1,P\},\{rct_2,P\}),true\})$$

Graph Transformation: See Figure 4.24.

Insert Random Choice Variation IRC_2 :

Description: Inserts a Random Choice and Flow Merge in sequence with an existing atomic task.

Policy syntax: *insert(RC_{new}, T, false)*

Grammar: *Insert::* $T \xrightarrow{(RC(\{rct_1,P\},\{rct_2,P\}),false)} T; RC(\{rct_1, P\}, \{rct_2, P\})$

Graph Transformation: See Figure 4.25.

Insert Random Choice Variation IRC_3 :

Description: Inserts a Random Choice and Flow Merge in parallel with an existing composite task.

Policy syntax: *insert(RC_{new}, CT, true)*

Grammar: *Insert::* $CT \xrightarrow{(RC(\{rct_1,P\},\{rct_2,P\}),true\}}$

$$FJ(2, \{CT, true\}, \{RC(\{rct_1, P\}, \{rct_2, P\}), true\})$$

Graph Transformation: See Figure 4.26.

Insert Random Choice Variation IRC_4 :

Description: Inserts a Random Choice and Flow Merge in sequence with an existing composite task.

Policy syntax: *insert(RC_{new}, CT, false)*

Grammar: *Insert::* $CT \xrightarrow{(RC(\{rct_1,P\},\{rct_2,P\}),false)} CT; RC(\{rct_1, P\}, \{rct_2, P\})$

Graph Transformation: See Figure 4.27.

4.4.3 Delete

Delete is a reconfiguration function that removes the workflow item from the current workflow instance whose identification is the given input. It can be applied to any workflow item, even one that has only been inserted into the control flow through a policy. Deleting a non-existing workflow item, or at least one that is not included in the control flow, has no effect. The literal translation of performing a delete is “do not execute this workflow item”.

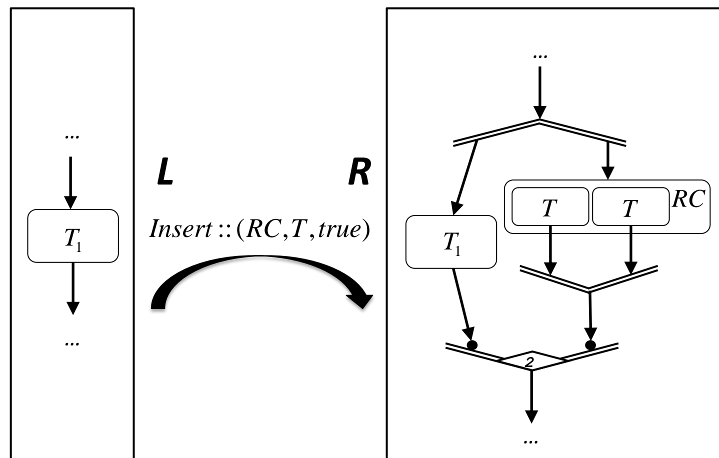


Figure 4.24: GT rule for insert Random Choice variation IRC_1.

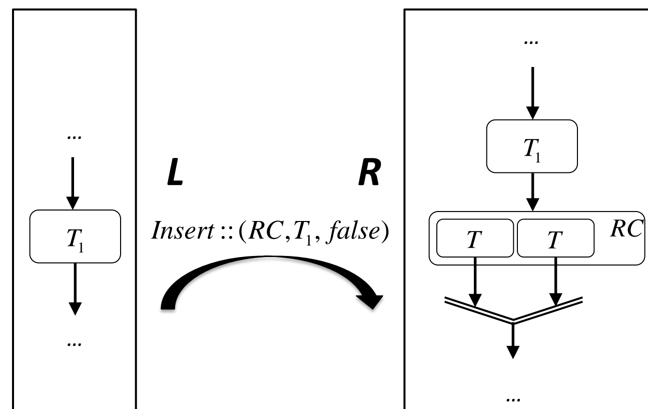


Figure 4.25: GT rule for insert Random Choice variation IRC_2.

A deletion can be of two forms: deletion of a task (atomic or composite), or deletion of an operator. Although there is a semantic difference between deleting an atomic task and deleting a composite task, we will see there is little conceptual difference as a workflow item can be either a single task or a sequence of items, including tasks and operators. Furthermore, there is little to conceptually differentiate decision operator deletions, Strict Preference deletions and Random Choice deletions. However, we include each scenario for completeness.

Since a task can have up to three outgoing control flow paths (for completion, abortion and failure), we must take into account the mapping of of the relevant paths now there exists a gap in the workflow. The following task on the completion path is mapped directly to the incoming task control flow path to ensure continuity. However, since the task no longer

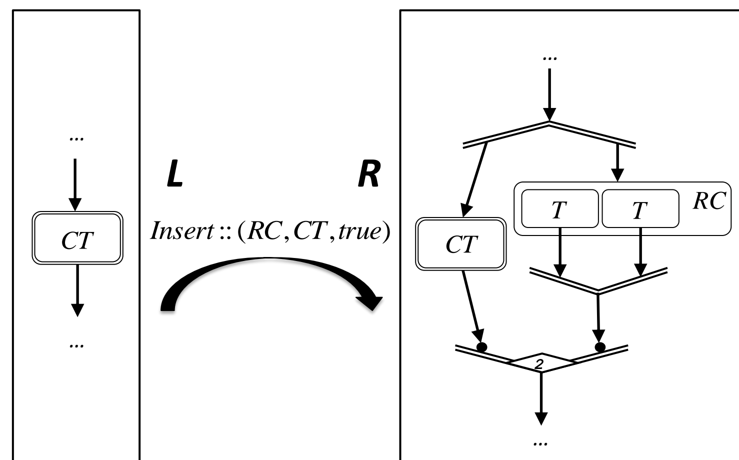


Figure 4.26: GT rule for insert Random Choice variation IRC_3.

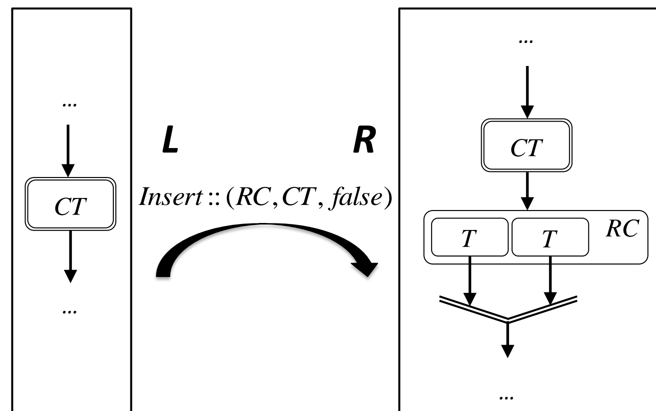


Figure 4.27: GT rule for insert Random Choice variation IRC_4.

exists, it can neither fail or be aborted. Therefore, these paths can be ignored since they can never be followed. It implies that if a workflow item would be executed exclusively if another task failed or was aborted, the item becomes redundant if that task was deleted.

We proceed to describe each of the variations of the Delete function, using policy syntax, grammar and graph transformation rules.

For clarity, the start and end points in the workflow have been included in L and R graphs. These are proceeded and preceded by, respectively, a "..." representing any other workflow item. Any deleted items remain inside the R graph, but are separated from the main control flow (i.e. they will not execute because they cannot be reached) and thus they faintly coloured to illustrate this.

Delete Task

Removing a task from the workflow is the most basic function available, suitably easier than inserting a task. The function detaches the specified task from its position inside the control flow and connects the former incoming and outgoing branches, thus creating one edge from the previous two.

For example, suppose a person applies for a a credit card. This normally involves a credit check and upon a positive result, a card is authorized. A variation to this process can be that if the application has a substantially high income (say over £100,000), then the credit check is not necessary. In that case, the process consists of solely authorizing the card. This is illustrated in Figure 4.28.

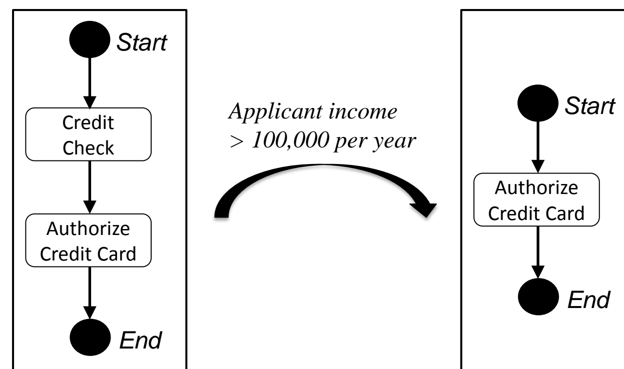


Figure 4.28: Credit check illustration.

The policy usage of this function is simple: *delete(t)*, where *t* is task ID. The grammar is $Delete :: T \xrightarrow{T} 0$, where 0 represents an empty state. Finally, the graph transformation rule is shown in Figure 4.29.

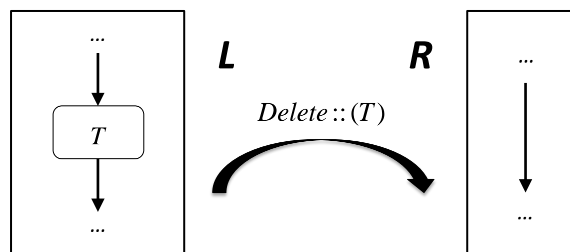


Figure 4.29: GT rule for deleting a task.

Delete Composite Task

Deleting a composite task is also straightforward, provided the composite task is already defined. Otherwise, the way to delete a composite task is either to define it first, then delete it, or by deleting every workflow item within the correct boundary. For simplicity, we assume that the composite task is always defined.

The policy syntax of this function is *delete(CT)*, where CT is composite task ID. The grammar is $Delete :: CT \xrightarrow{CT} 0$, where 0 represents an empty state. Finally, the graph transformation rule is shown in Figure 4.30.

All workflow items inside the composite task are detached from the workflow, but left attached to one another, i.e. the incoming and outgoing flows to and from the composite task are left "floating", whereas every connection inside the composite task remains. This facilitates the reuse of this composite task at a later stage.

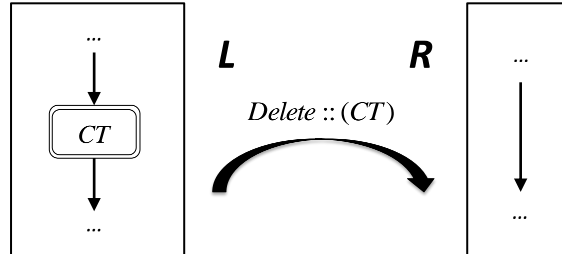


Figure 4.30: GT rule for deleting a composite task.

Delete Flow Split

Deleting a Flow Split operator includes deleting the actual operator plus its related Conditional Merge operator and every workflow item between the two. As with the composite task, all internal connections remain after deletion to facilitate reuse. The policy syntax of this function is *delete(FS)*, where FS is the unique name of the flow split. The grammar is $Delete :: FJ(m,\{P,\beta\},...,\{P,\beta\}) \xrightarrow{FJ(m,\{P,\beta\},...,\{P,\beta\})} 0$, where m is the number of mandatory branches required in the join, the pairs $\{P,\beta\}$ represent each outgoing branch and their

merging requirement (i.e. mandatory or optional) and 0 represents an empty state. The graph transformation rule is shown in Figure 4.31.

Although the policy syntax refers to the Flow Split operator by its own identifier, using the grammar we realize that the Flow Split is referenced through its Conditional Merge operator.

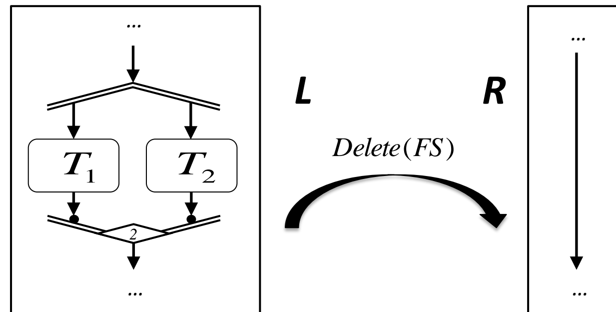


Figure 4.31: GT rule for deleting a Flow Split.

Delete Flow Junction

Deleting a Flow Junction follows from the deletion of a Flow Split. The policy syntax is *delete(FJ)*, where *FJ* is the unique ID of the operator. The grammar is *Delete ::* $\lambda^{?}T_1 : T_2 \xrightarrow{\lambda^{?}T_1:T_2} 0$, where T_1 and T_2 represent any (set of) workflow items inside the junction (including no item at all), and the graph transformation rule is shown in Figure 4.32.

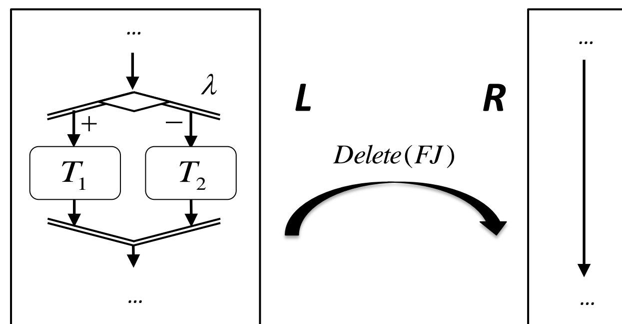


Figure 4.32: GT rule for deleting a Flow Junction.

Delete Strict Preference

This function is similar to deleting the flow junction operator. The policy syntax is *delete(SP)*, where *SP* is the unique name of the operator. The grammar is *Delete ::* $SP(\{spt_1, ct_1\}, \{spt_2, ct_2\}) \xrightarrow{SP(\{spt_1,ct_1\},\{spt_2,ct_2\})} 0$, where spt_1 and spt_2 are the task contents of the operator (although more may exist) and the respective workflow items (shown as "...") in the respective outgoing control flows of the tasks in the operator. The graph transformation rule is shown in Figure 4.33. Note that deleting the operator also deletes all tasks that are defined prior to the Flow Merge operator.

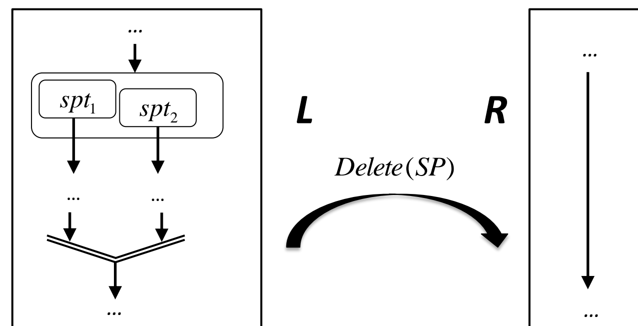


Figure 4.33: GT rule for deleting a Strict Preference.

Delete Random Choice

This function is similar to deleting the Flow Junction and Strict Preference operators. The policy syntax is *delete(SP)*, where *SP* is the unique name of the operator. The grammar is *Delete ::* $RC(\{rct_1, ct_1\}, \{rct_2, ct_2\}) \xrightarrow{RC(\{rct_1,ct_1\},\{rct_2,ct_2\})} 0$, where rct_1 and rct_2 are the task contents of the operator (although more may exist) and the respective workflow items (shown as "...") in the respective outgoing control flows of the tasks in the operator. The graph transformation rule is shown in Figure 4.34. Note that, as with Strict Preference, deleting the operator also deletes all tasks that are defined prior to the Flow Merge operator.

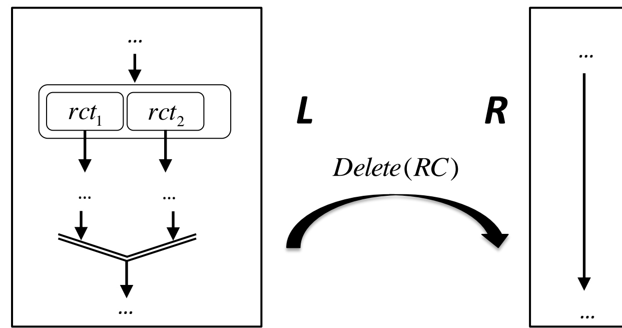


Figure 4.34: GT rule for deleting a Random Choice.

Exceptional Delete

For more complex workflows, multiple end points may be drawn as a shorthand way of combining a number of flows into a single workflow end point. In order to assist with this situation, StPowla allows a user to define a number of end points, whereas the workflow grammar requires just one end point. The implication for using the delete function is that potentially the situation may arise where a user deletes a scope (i.e. a sequence of workflow items) that includes an additional end point. By removing this, we would also be required to move the original flow split operator and the final merge operator, plus any workflow item contained between the two.

Instead, StPowla permits the user to delete a scope that includes an additional end point, without affecting the remainder of the workflow. Should the user wish to delete more items, they should write new policies.

4.4.4 Fail and Abort

The Fail function discards any potential output from the task being processed and diverts control flow onto an outgoing error flow. We recall that a task can have three outgoing control flows: 1) On completion, 2) On failure, 3) On abort. The Fail function simply diverts control flow to the fail path and the abort function does similarly to the abort path.

To illustrate situations when these functions may be necessary, consider an insurance

policy that is paid for through monthly premiums. Should the subject of the policy fall behind in payments, they will be sent a letter to inform them of the amount they are in arrears. Each letter may be accompanied by a phrase such as "*If you have paid in the last seven days, please excuse this letter*". Correctly interpreted, this phrase suggests that if postage of the letter takes 2 days, then there is a minimum 5 day processing period for the letter. Should a payment be received from the policy subject within this time, the action of sending the letter can be aborted.

More simply, the process of making coffee fails if there are no more instant coffee granules. A failure here could lead to making tea instead. A separate condition could be that in the summer, if there is no coffee, fruit juice could be consumed instead.

Both functions may optionally call a service's cancel or rollback function, should one exist. However, since the user is naive about service usage, they do not know about this particular step.

A graph transformation rule is not applicable for these functions since there is no structural change to the workflow. However, there is still the possibility for the user to manipulate the control flow. For example, in Object Oriented languages such as Java and C#, the concept of Exceptions is sometimes used for more than just error handling. It can be used as a means of defining alternative flows depending on the outcome of a particular situation.

Likewise, the workflow construction grammar is not affected either.

In terms of policy syntax, the functions can be invoked through *fail()* and *abort()*, respectively. The functions can only be used in policy action sections and thus apply only to the task specified. By implication, these functions cannot be applied to operators. However, they can be applied to composite tasks, where the active atomic task (if any) is the one that is failed or aborted.

4.4.5 Block

The block function can be applied to any atomic task. Each task has an attribute *ready* indicating whether or not the task itself is ready to be executed. If the value is *true* and the task is reached in the control flow (i.e. the task is activated), then it proceeds to perform its normal execution. If the value is *false* and the task is reached in the control flow, then it waits until the value changes to *true* or that an abort or fail function is called on that task.

The purpose of block is to place a hold on (i.e. delay) the execution of tasks. For example, in the supplier example, there is an obvious requirement that if the supplier does not have enough stock to fulfil the order, they must delay the collection of items and in the meantime order new stock.

The policy syntax is *block(t,p)*, where *t* is the name of the task to be blocked and *p* is the continuation condition. The function can be called from any policy, not just a policy that applies to that specific task. In the supplier example, it could be demonstrated as follows:

```
policy Pause Order is
appliesTo ReceiveOrder
     when taskCompleted
       if global.StockLevel < Order.Quantity
       do block(CollectItems, global.StockLevel > Order.Quantity)
```

This policy assumes a global variable *StockLevel*, holding the current stock level of items, and an entity *Order* with attribute *Quantity*, which holds the quantity of items requested in the order. For simplicity, we assume that the supplier only sells one unique item.

To solve this problem, we could add another policy as follows:

```
policy Notify Delay is
appliesTo ReceiveOrder
```

```
    when taskCompleted
      if CollectItems.ready == false
      do   insert(SendDelayNotice, this, false)
         andthen
           insert(GetStock, SendDelayNotice, true)
```

This policy, written after the previous policy, is applied to the same *ReceiveOrder* task. The triggering event is when the task completed and the condition is when the *CollectItems* task is not ready (i.e. it is blocked) to execute. In this situation, a composite action is performed where firstly a delay notice is sent to the client (task *SendDelayNotice*) after the current task and secondly a task is inserted in parallel to that task with the purpose of ordering new stock (*GetStock*).

A third policy is written as follows, to unblock the original task after *GetStock* has completed successfully.

```
policy Resume Order Process is
appliesTo GetStock
    when taskCompleted
      do CollectItems.ready := true
```

For completeness, we can include a policy on the *SendDelayNotice* task that attempts to abort if the *GetStock* task completed first.

```
policy Abort Delay Notice is
appliesTo SendDelayNotice
    when taskStarted
      if CollectItems.ready == true
      do abort()
```

The graph transformation rule is shown in Figure 4.35, where rather than a workflow graph in L and R, these graphs are shown as pseudo-UML class diagrams, with the

task/class name shown, with a variable *ready* and its value shown underneath. In addition, example *G* and *H* workflows are shown wherein attributes of tasks are displayed.

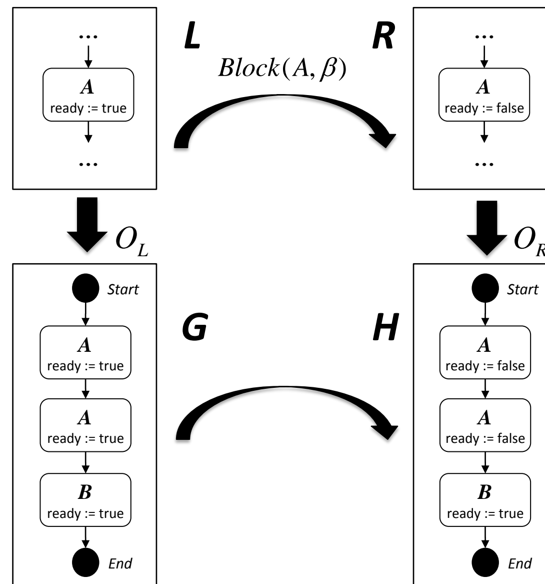


Figure 4.35: GT rule for block.

4.5 Summary

In this chapter, we have identified, described and defined using policy syntax, logical syntax and graph transformation rules a set of functions that can reconfigure a workflow. These functions are insert, delete, fail, abort and block. Although superficially simple, the rules require careful analysis, especially with respect to the context in which they will execute.

However, even with only this small set of reconfiguration functions available, we can perform wholesale systematic modifications to a particular workflow instance, without losing the structure of the core workflow. Insertions allow extra functionality to occur inside the process whilst deletions allow current functionality to be removed. Causing a task to fail or abort can change the control flow path inside the workflow. Blocking allows the execution of particular functionality to wait for some other execution to start, or finish, first.

Inserting new functionality is not a trivial matter however. To insert a task into the workflow, one must consider four variations. If that task is a composite task, there are four further variations. When considering the insertion of operators, the correctness of the workflow is only maintained correctly if a flow divergence operator (i.e. a Flow Split, Flow Junction, Strict Preference or Random Choice) is accompanied by an appropriate join operator. This means that for each item or set of workflow items to be inserted have a single incoming control flow point and a single outgoing control flow point.

Deleting workflow items works on a similar principle. The workflow correctness cannot be compromised by removing a divergence operator whilst leaving its related convergence operator. As such, either the entire construct must be deleted or it must not. Therefore, to reuse components inside the workflow, one must first insert a copy of each required item elsewhere in the workflow and construct the new control flow, before then deleting the existing operator.

Blocking, Fail and Abort provide means to control how the workflow is executing, rather than just the process flows, thus providing a second mechanism for workflow control.

Each of the reconfiguration functions has been described in three ways:

1. How the reconfiguration function is invoked within a policy;
2. The formal construction of the reconfiguration function;
3. The graph transformation rule to illustrate the function's effects and to facilitate the combination of multiple functions.

Through these reconfiguration functions, a workflow can be constructed purely from an empty state if required. However, and more likely, they can be used to provide powerful modifications to the executing workflow instance without having to undergo changes to the workflow itself or the workflow management system.

We proceed in the next chapter to formalize these reconfiguration functions using the SENSORIA Reference Modelling Language.

Chapter 5

From StPowla Processes to SRML Models

All he'd wanted were the same answers the rest of us want. Where did I come from? Where am I going? How long have I got? (Blade Runner).

5.1 Introduction

In Chapter 3 we introduced StPowla as a combination of workflows, policies and Service Oriented Architecture (SOA). In Chapter 4, we presented a set of reconfiguration functions that could be used in policies to affect the structure of the executing workflow instance. In this chapter, we provide more substance to the relationship between the business domain (i.e. the combination of workflows and policies) and the technical domain (i.e. Services). We do this through providing an encoding from StPowla processes to SRML models [1, 29].

Service Oriented Computing (SOC) is a paradigm for developing software systems as the

composition of a number of services. Services are loosely coupled entities that can be dynamically published, discovered and invoked over a network. A service is an abstract resource whose invocation triggers a possibly interactive activity (i.e. a session) and that provides some functionality meaningful from the perspective of the business logic [49]. A SOA allows services with heterogeneous implementations to interact relying on the same middleware infrastructure. Exposing software in this way means that applications may outsource some functionalities and be dynamically assembled, leading to massively distributed, interoperable and evolvable systems.

The engineering of service oriented systems presents novel challenges, mostly due to this dynamicity [100]. In this chapter we use SRML models, which focus on the modelling of orchestrations. An orchestration is the description of the executable pattern of service invocations/interactions to follow in order to achieve a business goal.

Put simply, SRML allows us to understand the behaviour of the system that implements StPowla. The pattern of execution, i.e. the orchestration, is shown through a series of messages, or *transitions*. These transitions demonstrate the system state changes and thus the order of execution of the resulting effects.

In reality, we need to address both the business perspective in which StPowla operates, and the technical perspective in which SRML operates. Both have their own challenges, as we have seen with StPowla already. We must also address the relationship between the two perspectives. In the light of this, we discuss the relationship between the two modelling languages for service oriented systems developed in the context of Sensoria.

SRML is a high-level modelling language for SOAs whose goal is "to provide a set of primitives that is expressive enough to model applications in the service-oriented paradigm and simple enough to be formalised" [29]. SRML aims to represent the various foundational aspects of SOC (e.g. service composition, dynamic reconfiguration, service level agreement, etc.) within one integrated formal framework. A declarative semantics has been provided in [2, 58] that maps SRML to mathematical domains that

make precise the meaning of the different constructs made available in SRML. In particular, [2] provides a formal computational model for SRML which is being mapped into a logic adapted from μUCTL, a formalism being developed within Sensoria for supporting qualitative analysis [34].

We borrow the description of SRML from one of its founding authors:

> SRML is a modelling language for service-oriented systems developed by the IST-FET-GC2 Integrated Project SENSORIA. SRML operates at the higher levels of abstraction of business modelling, i.e. it provides a number of semantic modelling primitives that are independent of the languages and platforms in which services are programmed and executed. In particular, SRML abstracts from the typical mechanisms made available by service-oriented middleware such as sessions and event/message correlation, as well as the brokers that are responsible for the discovery and binding of services. A formal computation and coordination model was developed for SRML over which qualitative and quantitative analysis techniques were defined using the UMC model checker and the PEPA stochastic analyser. An algebraic semantics was also developed for the run-time discovery, selection and binding mechanisms. Finally, methodological aspects of engineering business services and activities were investigated, which were supported through extensions of use-case and message-sequence diagrams and tested over a number of case studies.[1]

One can say that SRML is complete in its expressive power with respect to the systems we intend to model. While expressivity is clearly an issue to computer scientists, usability is the more important factor for business analysts. StPowla addresses usability partly in making use of graphical notations and more crucially in being modular in that the basic workflow and the policies capturing variability are kept separate while SRML is

[1] `http://www.cs.le.ac.uk/srml/`

essentially flat in that it merges both into the same description.

The encoding of StPowla into SRML on the one hand provides a formal framework to StPowla. StPowla workflows can be then represented as SRML models and can either be analysed alone or as part of more complex modules, where they are composed with other SRML models with heterogeneous implementations (e.g. SRML models extracted from existing BPEL processes [24]).

A second reason for the encoding is providing a higher layer to the modelling of orchestrations in SRML that includes a process-based approach to the definition of a workflow, a separate view of policies, that had not been yet considered in SRML, and the inter-relation between workflow and policies.

Having already provided a substantial introduction to StPowla in this thesis, we proceed directly to an introduction to SRML. We then provide an encoding from StPowla notation to SRML models. Following this, we incorporate policies and significantly the reconfiguration functions in order to provide an advanced encoding to SRML. In the following chapter, we provide a case study for StPowla and include the methodology for encoding it to SRML models.

5.2 SRML Foundational Concepts

In SRML composite services are modelled through *modules*. A module declares one or more components (that are tightly bound and defined at design time), a number of requires-interfaces that specify services (that need to be provided by external parties), and (at most) one provides-interface that describes the service that is offered by the module. A number of wires establish interaction protocols among the components and between the components and the external interfaces. Figure 5.1 shows the SRML module ProcurementService which includes one provides-interface CR (i.e. the interface of the service

provided to the customer), one requires-interface (i.e. the interface of the service that we will discover at run-time) and two components: BP and PI orchestrate the interactions among the parties.

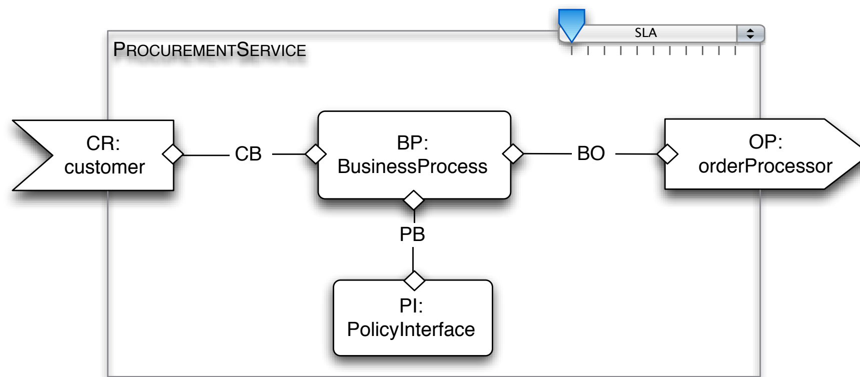


Figure 5.1: The structure of a SRML module for the procurement service example

In layman's terms, CR shows the system offers a service to an external consumer. OP shows that this ProcurementService module requires use of an external service, which it refers to as OP. BP and PI are internal programs (perhaps services themselves) that co-ordinate this module. The former manages the execution of the core process and the latter manages variability aspects.

The internal nodes of a SRML module can reside at three different layers: top layer, service oriented layer and bottom layer. These layers are architectural abstractions that reflect different levels of organisation and change. Each layer uses the layer underneath. The top layer uses the service-oriented layer to achieve a business goal. For example, an application could be designed for the same organisation that intends to use it and not for being published as a service. These applications are known as *activities*. The creation of an activity instance is triggered by a node that belongs to the top layer and not by a provides-interface.

The service oriented layer uses the bottom layer which typically includes entities which are persistent as far as the life cycle of the activities is concerned, and can be shared by multiple instances of the same activity (e.g. a database shared by all the instances of a

service).

Here we focus on SRML modules where nodes reside only at the service-oriented layer. This simplification is done without loss of generality since it does not have any influence on the encoding. In fact, StPowla does not make any distinction about the type of entity that performs a task (e.g. dynamically discovered service vs persistent resource). The interested reader can refer to [11] for more details on the layered structure of SRML modules.

Components, external interfaces, wires and interfaces of the different layers are specified in terms of *Business Roles*, *Business Protocols*, *Interaction Protocols* and *Layer Protocols* respectively. The specifications define the type of the nodes. Each specification provides a slightly different style of behavioural description. A business role defines an execution pattern involving the interactions that it declares in its signature, what we call an orchestration. Business protocols provide a set of properties that abstracts from details of the executable process implemented by the orchestration (e.g. the local state) and describe the behaviour that can be expected of the service (in case of provides-interface) or specify the behaviour that is expected (in the case of requires-interface) of the external party.

The interaction protocols define a collection of properties that establish how the interactions are coordinated, which may include routing events or transforming sent data to the format expected by the receiver. However, each language has been captured in the computational model presented in [2], which defines the activity of a configuration of SRML components in terms of transition systems where transitions represent the sending, receiving and processing of events by the entities involved in the business activity. The logic $UCTL$ [34] is being used to reason about such transition systems. The aim is to provide (1) a notion of correctness for service modules (i.e. the properties of a provides-interface are entailed by the body of a module, assuming the properties described in the requires/uses-interfaces), (2) a way of formalising the matching of provides/requires-interfaces, and (3) a means for validation of activity and service design.

In this chapter we provide an encoding to derive, from a business process specified in StPowla, an SRML component that we call BP, of type BusinessProcess and a second component PI, of type PolicyInterface that is connected to BP and represents the interface through which it is possible to trigger policies that modify the control flow. PI supports the set of interactions used to trigger a workflow modification in the component BP. Figure 5.1 illustrates the structure of the SRML module representing the workflow and policies in the procurement example described earlier in this section.

Components are instances of business roles specified in terms of (1) the set of supported interactions, and (2) the way in which the interactions are orchestrated. The next section provides an overview of business roles. The overview will not include the other types of SRML specification as they are not required for the encoding.

5.2.1 Transitions

In the mapping of StPowla to SRML, we are concerned with the orchestration and to see how a StPowla workflow and set of policies interact from the perspective of SRML. We are mainly interested in transitions - the interactions between components that implement the required functionality of the workflow and policies.

A transition can be defined at different levels: (1) at the module level, giving a high level view of the workflow execution; and (2) at the inter-component level. The general syntax is as follows:

```
transition ...
    triggeredBy trigger
    guardedBy guard
    effects effects
```

In the above transition, we are essentially defining a state change of the system using

an ECA-style rule. The triggering event is defined in the **triggeredBy** clause. There could be multiple triggers, composed using the ∨ symbol (noting that each trigger can independently effect the transition). The triggers do not imply that either an event has happened, but that the system has enabled signals to indicate that those events may now occur.

The **guardedBy** clause is the condition that must be satisfied for the transition to occur.

The **effects** clause defines what should happen as a result of the transition being execution. There could be one effect or a set of effects. Where two or more triggers are present, the symbol ⊃ joins the effects to the trigger. Where the effects syntax a ⊃ b occurs, this means that when trigger a occurs during the execution, effect b happens. Effect b could be a composition of effects c and d, which are joined through the symbol SRML (i.e. the full syntax would be a ⊃ c ∧ d).

In the transition, it is possible for us to define effects based on the trigger. Although in this example there are two triggers, we have defined two separate sets of effects with one for each trigger. The symbol ⊃ joins the effects to the trigger, whereas the symbol ∧, which starts a new line of effects and its own a ⊃ b syntax, separates the two sets of effects. Therefore if trigger a led to effect b, and effect e led to effect f, we could encode that in the same transition as follows:

```
transition ...
    triggeredBy a ∨ b
    guardedBy ...
    effects
        a ⊃ b
        ∧ e ⊃ f
```

5.2.2 Business Roles: the Interactions

In a SRML module each component, including the provides-interfaces and requires-interfaces where they exist, send and receive messages to one another as part of the orchestration. The component that we focus on at any one time is referred to as the *party* and the other component that it is interacting with as the *co-party*.

SRML supports asynchronous two-way conversational interactions: **s&r** denotes interactions that are initiated by the co-party, which expects a reply, **r&s** denotes interactions that are initiated by the party, which expects a reply from its co-party. SRML supports also asynchronous one-way and synchronous interactions that are not discussed here as they are not required for this work.

Having understood the conversational interactions, we can now specify a set of interactions supported by PolicyInterface, with each interaction corresponding to one of the StPowla functions in Table 4.1. The business role BusinessProcess supports the complementary interactions (i.e. **r&s** instead of **s&r**) plus other interactions that occur with the external parties. Each interaction can have 🔔-parameters for transmitting data when the interaction is initiated and ✉-parameters for carrying a reply. The index *i* represents a key-parameter that allows us to handle occurrences of multiple interactions of the same type (as in SRML every interaction event must occur at most once). In this case, we allow PI to trigger more instances of policy functions of the same type.

5.2.3 Business Roles: the Orchestration

The way the declared interactions are orchestrated is specified through a set of variables that provide an abstract view of the state of the component, and a set of transitions that model the way the component interacts with its co-parties. For instance, the local state of the orchestrator is defined as follows:

```
BUSINESS ROLE PolicyInterface is
_____________________________________
INTERACTIONS
        s&r delete[i:natural]
                🔔 task:taskId
        s&r insert[i:natural]
                🔔 task:taskId
                   newTask:taskId
                   c:condition
        s&r block[i:natural]
                🔔 task:taskId
                   c:condition
        s&r fail[i:natural]
                🔔 task:taskId
        s&r abort[i:natural]
                🔔 task:taskId


    local

    start[root],start[x],start[ro],...:boolean, ...

    state[root],state[x],state[ro],...:[toStart,running,exited]
```

A module can define an initialisation condition for the each component. For example, the module ProcurementService may define the following initial state for the component CR:

```
    start[root]=true

    ∧ start[x]=start[ro]=...=false

    ∧ state[root]=state[x]=state[ro]=...=toStart ∧ ...
```

Similarly, a termination condition may specify the situations in which the component has terminated any activity. The behaviour of components is described by transition rules. Each transition has a name and a number of other features:

```
    transition policyHandlerExample

        triggeredBy samplePolicy🔔[i]

        guardedBy state[samplePolicy🔔[i].task] = toStart
```

```
effects policy[samplePolicy🔔[i].task]' ∧ ......
sends samplePolicy✉[i]
```

triggeredBy is a condition, typically the occurrence of a receive-event or a state condition, which triggers the execution of the transition. In the example we engage the *policyHandlerExample* transition when we receive the initiation of the interaction *samplePolicy* (i.e. *samplePolicy*🔔[*i*]).

guardedBy is a condition that identifies the states in which the transition can take place. For instance, the *policyHandlerExample* transitions should only be taken when the involved task is in state *toStart* (i.e. is not in execution and it has not been executed yet). The involved task is identified by the parameter *task* of the interaction *samplePolicy* (i.e. *samplePolicy*🔔[*i*].*task*).

effects concern changes to the local state. We use *var*′ to denote the value the state variable *var* has after the transition.

sends is a sentence that describes the events that are sent and the values taken by their parameters. In the example we invoke the *samplePolicy* reply event (i.e. *samplePolicy*✉[*i*]) to notify of the correct management of the policy.

5.2.4 Constraints for Service Level Agreement in a SRML Module

Although not important to this work, we also briefly discuss the Service Level Agreement aspect of SRML modules, to go alongside the same concept from StPowla (see Section 3.6).

SRML offers primitives for modelling the dynamic aspects concerned with session management and service level agreement, which together we call configuration policies. The *external configuration policy* concerns the constraints that the process of discovery, negotiation and binding must satisfy to establish service level agreements (SLA) with service

providers. The external configuration policy models an orthogonal aspect with respect to the orchestration. Specifically, it defines a set of non-functional properties to be considered when, in a specific point of the orchestration process, the run-time discovery of an external service for outsourcing the execution of a specific task is required.

SRML uses an algebraic approach developed in [81] for constraint satisfaction and optimization. The following example uses a constraint system where the degree of satisfaction has fuzzy values, i.e. it takes values in the interval [0, 1].

```
EXTERNAL POLICY
________________________________

SLA VARIABLES
    OP.LOCATION, CR.LOCATION
CONSTRAINTS
    Closeness is <{OP.LOCATION, CR.LOCATION},def2> s.t.
        if distance(OP.LOCATION, CR.LOCATION) < 50 then def2(n,m)=1,
        otherwise def2(n,m)=500/n
```

In order to define the constraints that we wish to apply to the module ProcurementService, we use the SLA variables OP.LOCATION and CR.LOCATION which are the locations of the order processor and customer, respectively. We define only one constraint Closeness, which minimises the distance between the customer CR and the order processor OP. The best degrees of satisfaction are when the distance is less than 50 miles. Otherwise they are inversely proportional to the distance. The function distance returns the distance between two locations.

For each potential order processor (i.e. the service, among the published ones, whose provides-interface matches with OP, of type OrderProcessor), the set of constraints has to be solved. The solution assigns a degree of satisfaction to each possible tuple of values for the SLA variables. Negotiation in this framework consists in finding an assignment that maximizes the degree of satisfaction. Hence, the outcome of the negotiation between

ProcurementService and the potential partner is any tuple that maximizes the degree of satisfaction. Selection then picks a partner with a service level agreement that offers the best degree of satisfaction.

5.3 Basic Control Flow Encoding

In this section we present an encoding from the control constructs (i.e. notation operators) of StPowla to SRML orchestrations. Our focus is on the control constructs and we abstract from the interactions of the service and from the semantics of the simple activities of the workflow tasks.

StPowla represents a business process as the composition of a number of tasks, either simple (e.g. interactions with services) or complex (e.g. coordinating other tasks by executing them in sequence, parallel, etc.). In SRML we associate an identifier, of type *taskId*, to any task. We denote with T the set of all the task indexes in the workflow schedule.

For each task identifier x we define the following local variables, used to handle the control flow and coordinate the execution of the tasks:

- $start[x]$ is a boolean variable that, when true, triggers the execution of x;
- $done[x]$ is a boolean variable that signals the successful termination of x and triggers the continuation of the workflow schedule;
- $fail[x]$ is a boolean variable that signals the termination with failure of x and triggers the failure handler.

In general, the next activity in the control flow is executed when the previous one completes successfully. In case of task failure the output control flow follows a pre-defined path specific to the failure of the task. If such a path does not exist, and no other failure

handler has been defined, the workflow continues as normal. We leave the specification of the failure handling mechanisms in SRML as a future work.

The Strict Preference and Random Choice operators, that try a number of alternative tasks until one terminates with success, handle the failure signal directly, within the workflow. The scope construct can be extended in the future to support failure handling, as discussed later.

We introduce in section 5.4 a set of transitions, as a part of the orchestration of BP that models the policy handler. The policy handler has the responsibility to enact the reconfigurations of the workflow control flow specified by the policies triggered by PI. The policy handler blocks the normal flow by setting the variable $policy[x] = true$, where x is the identifier of the first task involved in the modification. The variable $policy[x]$ is a guard to the execution of x. We will describe the policy handler more later in this chapter, by now it is important to know that when a policy function has to be executed on a task, the task has to be blocked. It is responsibility of the policy handler to reset the flow of execution.

Some policies can be applied only on running processes (e.g. abort) and some others only on tasks that have not started yet (e.g. delete). We define a local variable $state[x]$ for every task identifier x which identifies the state of the execution of the transition associated to x by taking one of the following values: *toStart* (i.e. the execution of the task has not started yet), *running* (i.e. the task is in execution) and *exited* (i.e. x has terminated).

The initialisation conditions for the module set, for each task identifier x, $state[x] = toStart$, see for example the initialisation condition for ProcurementService presented in Section 5.2.2. The state variable $state[x]$ is be used to ensure that policies act on a task in the correct state of execution (i.e. the deletion of task x can be performed only if $state[x] = toStart$).

We consider the simple tasks as black boxes: we are not interested in the type of activity that they perform but only on the fact that a task, for example task x, is activated by

$start[x]$, signals its termination along either $done[x]$ or $failed[x]$ and notifies its state along $state[x]$.

The execution of the workflow is started by a special transition *root* that sets $start[x]$ =*true* where x is the first task in the workflow schedule. The local variables are initialised as follows: $\forall i \in T \setminus root, start[i] = false \wedge start[root]$ =*true*, $\forall i \in T, done[i] = failed[i] =$ $policy[i]$ =*false* and $\forall i \in T, state[i]$ =*toStart*.

It follows the encoding of the workflow template $start; P; end$ where P is associated to the task identifier x:

```
transition root
    triggeredBy start[root] ∨ done[x]
    guardedBy ¬ policy[root]
    effects
    start[root]⊃ ¬ start[root]’ ∧ state[root]’=running ∧ start[x]’
    ∧ done[x] ⊃ ¬ done[x]’ ∧ done[root]’ ∧ state[root]’=exited
```

The guard of transition *root* ensures the execution of the transition only if no policy has been triggered on task *root* (i.e. $policy[root]$ is false). According to the trigger, *root* is executed twice:

1. at the beginning of the workflow (recall that the initialisation condition of ProcurementService includes the assignment $start[root] = true$). The transition in this case has the following effects: (1) disabling the the triggering condition of *root* (i.e. $start[root]$ is set to $false$), (2) setting the state of task *root* to *running* and (3) triggering the transition for task x by setting $start[x]$ to *true*.
2. when task x terminates (i.e. $done[x] = true$). The transition in this case has the following effects: (1) disabling the termination signal for x is disabled (i.e. $done[x]$ is set to $false$), (2) enabling the termination signal for *root* i, and (3) setting the state variable for *root* to *exited*.

5.3.1 Sequence

The sequence operator $P_1; P_2$ first executes P_1 and, after the successful termination of P_1, executes P_2. We remark that failures are not handled in this thesis and will be addressed in the future.

The encoding of the sequence construct in SRML is as follows. The sequence is encoded in the following SRML transition, with task identifier x, which triggers the execution of the first task, with task identifier $p1$, then collects the termination signal from $p1$ and triggers the execution of the second subprocess, with task identifier $p2$:

```
transition X
    triggeredBy start[x] ∨ done[p1] ∨ done[p2]
    guardedBy ¬ policy[x]
    effects
    start[x] ⊃ ¬ start[x]’ ∧ state[x]’=running ∧ start[p1]’
    ∧ done[p1] ⊃ ¬ done[p1]’ ∧ start[p2]’
    ∧ done[p2] ⊃ ¬ done[p2]’ ∧ done[x]’ ∧ state[x]’=exited
```

Transition X is executed three times:

1. when $start[x]$ is true. The transition in this case has the following effects: (1) disabling the triggering condition $start[x]$, (2) changing the state of task x to *running* and (3) enabling the triggering condition $start[p1]$.

2. when $done[p1]$ is true (i.e. after $p1$ has been executed). The transition in this case has the following effects: (1) disabling the termination signal $p1$ and (2) enabling the triggering condition $start[p2]$.

3. when $done[p2]$ is true. The transition in this case has the following effects: (1) disabling the termination signal for $p2$, (2) enabling the termination signal for x and (3) setting the state of task x to *exited*.

5.3.2 Flow Junction and Flow Merge (XOR)

The Flow Junction and Flow Merge operator $\lambda^{?}P_1 : P_2$ consists of the combination of the Flow Junction, that diverts the control flow down one of two branches P_1 and P_2, represented by the task identifiers $p1$ and $p2$, respectively, according to a condition λ, and the Flow Merge of a number of flows where synchronisation is not an issue. The Flow Junction and Flow Merge are encoded into the following SRML transition:

```
transition X
    triggeredBy start[x] ∨ done[p1] ∨ done[p2]
    guardedBy ¬ policy[x]
    effects
    start[x] ⊃ ¬ start[x]' ∧ state[x]'=running
                   ∧ (λ ⊃ start[p1]') ∧ (¬ λ ⊃ start[p2]')
    ∧ done[p1] ⊃ ¬ done[p1]' ∧ done[x]' ∧ state[x]'=exited
    ∧ done[p2] ⊃ ¬ done[p2]' ∧ done[x]' ∧ state[x]'=exited
```

Transition X is executed twice:

1. when *start*$[x]$ is true. The transition in this case has the following effects: (1) disabling triggering condition *start*$[x]$, (2) setting the state of x to *running* and (3) triggering either $p1$ or $p2$ depending on the condition λ.

2. when either *done*$[p1]$ or *done*$[p1]$ is true (either $p1$ or $p2$ was executed). The transition in this case has the following effects: (1) disabling the termination signal for $p1$ or $p2$, (2) enabling the termination signal of x and (3) setting the state variable of x to *exited*.

5.3.3 Flow Split and Conditional Merge (AND)

The Flow Split and Conditional Merge operator $FJ(m, \{P_1, \mathcal{B}_1\}, \ldots, \{P_n, \mathcal{B}_n\})$ consists of the combination of the Flow Split, that splits the control flow over many branches, and the Conditional Merge, that synchronises two or more flows into one. The value of m, that is statically determined, represents the minimum number of branches that have to be synchronised. Furthermore, any branch is associated to a boolean $\mathcal{B}_i$ that determines whether the $i-th$ branch is mandatory in the synchronisation.

The encoding is as follows. Let S be the set, with cardinality n, of the task indexes associated to the branches of the Split/Merge. Let the identifiers for the subtasks of x to range over $p1, \ldots, pn$. Let N be the set of indexes of the necessary tasks and $m \in \mathbb{N}$ be the minimum number of branches that have to be synchronised. We assume that $0 \leq m \leq |N|$. The complex join is encoded in the following SRML transition, where $Kcomb$ is the set of $(m - |N|) - subsets$ of $S \setminus N$.

```
transition X
    triggeredBy start[x] ∨ (∧_{i∈N}done[pi]∧(∨_{K∈Kcomb}(∧_{k∈K}done[pj])))
    guardedBy ¬ policy[x]
    effects
    start[x] ⊃ ¬ start[x]' ∧ state[x]'=running ∧_{i∈[1,...,n]} start[pi]'
    ∧ ¬ start[x] ⊃ done[x]' ∧ state[x]'=exited ∧_{i:[1..n]}(¬ done[pi]')
```

The transition above is parametric with respect to N and K_{comb} in order to model the general case. In a real workflow schedule the general construct would be instantiated (at design-time) and the parameters in the conjuction/disjunctions would disappear (e.g. if $N = \{1, 2\}$ the term $\wedge_{i \in N} done[pi]$ becomes $done[p1] \wedge done[p2]$.

Transition X is executed twice:

1. when $start[x]$ is true. The transition in this case has the following effects: (1) disabling the triggering condition $start[x]$, (2) setting the state of task x to *running*

and (3) enabling the triggering condition for each sub-task i by setting $start[i]$ to $true$.

2. in case of successful termination of all the necessary subtasks (i.e. $\wedge_{i\in N} done[pi]$) and of a number of tasks greater or equal to m (i.e. $\vee_{K\in Kcomb}(\wedge_{k\in K} done[pj])$). The transition in this case has the following effects: (1) enabling the termination signal for x, (2) setting the state of x to *exited* and (3) disabling the successful termination of all the subtasks.

5.3.4 Strict Preference

The strict preference $SP(P_1, \ldots, P_n)$ attempts the tasks $P_1, \ldots, P_n$ one by one, in a specific order, until one completes successfully. In this case, with no loss of generality we consider the tasks ordered by increasing index numbers.

The Strict Preference is encoded in the following SRML transition:

```
transition X
    triggeredBy  start[x] ∨i:[1..n](done[pi] ∨ failed[pi])
    guardedBy ¬ policy[x]
    effects
    start[x] ⊃ ¬ start[x]' ∧ state[x]'=running ∧ start[p1]'
    ∧i:[1..n−1]failed[pi] ⊃ ¬ failed[pi]' ∧ start[p(i+1)]'
    ∧ failed[pn] ⊃ ¬ failed[pn]' ∧ failed[x]' ∧ state[x]'=exited
    ∧ ∨i:[1..n]done[pi] ⊃ done[x]' ∧  state[x]'=exited ∧i:[1..n]¬ done[pi]'
```

Transition X is executed a number of times in the following cases:

1. when the task x is triggered. The transition in this case has the following effects: (1) disabling the triggering condition $start[x]$, (2) setting the state of x to *running* and (3) triggering the first sub-task $p1$ of x.

2. when any of the tasks *pi* terminates with failure ($failed[pi] = true$) or success ($done[pi] = true$):

   - If the task failed (i.e. $failed[pi] = true$) and it was not the last task *pn*, then the transition has the following effects: (1) disabling the termination variable of *pi* and (2) enabling the next task by setting $start[p(i+1)]$ to true.
   - If the last task failed ($failed[pn] = true$) then the transition has the following effects: (1) disabling the signal of failed termination of *pn*, (2) enabling the signal $failed[x]$ of failed termination for *x* and (3) setting the state of *x* to *exited*.
   - If any of the sub-tasks terminated successfully ($done[pi] = true$) then the transition has the following effects: (1) enabling the signal of successful termination for *x*, (2) setting the state of *x* to *exited* and (3) disabling the successful terminations of all the sub-tasks.

5.3.5 Random Choice

The random choice $RC(P_1, \ldots, P_n)$ attempts the tasks $P_1, \ldots, P_n$ simultaneously and completes when one completes successfully. The random choice is encoded in the following SRML transition:

```
transition X
    triggeredBy start[x] ∨i:[1..n](done[pi])∨ (∧i:[1..n](failed[pi]))
    guardedBy ¬ policy[x]
    effects
        start[x] ⊃ ¬ start[x]' ∧ state[x]=running ∧i:[1..n]start[pi]'
        ∧ (∧i:[1..n]failed[pi]) ⊃ failed[x]'
                ∧ state[x]'=exited ∧i:[1..n]¬ failed[pi]'
        ∧ (∨i:[1..n]done[pi]) ⊃ done[x]' ∧ state[x]'=exited
                ∧i:[1..n](¬ done[pi]' ∧ ¬ failed[pi]')
```

Transition X is executed a number of times, in the following cases:

1. when the task x is triggered. The transition in this case has the following effects: (1) disabling the triggering condition $start[x]$, (2) setting the state variable of x to *running* and (3) enabling the triggering condition of all the sub-tasks.

2. when any of the tasks pi terminates with success (i.e. $done[pi] = true$). The transition in this case has the following effects: (1) enabling the successful termination of x, (2) setting the state of x to *exited* and (3) disabling the successful and faulty terminations for all the sub-processes.

3. when all the sub-tasks pi terminates with failure (i.e. $failed[pi] = true$). The transition in this case has the following effects: (1) enabling the faulty termination of x, (2) setting the state of x to *exited* and (3) disabling the faulty terminations for all the sub-processes.

5.3.6 Scope

The scope construct *Scope(P)*, where P is associated to the task identifier y behaves similarly to the root process.

Within larger business processes it often makes sense to group parts of the process together as these are controlled by one division in the company or the tasks are more intrinsically linked together. We represented scopes separately, even though at the moment little is done with them. The scope construct will in the future be extended to include notions of compensation and fault handling (that is they will form a way to express long running transactions), however this aspect is beyond the scope of this thesis. An idea is to support the mechanism of fault handling by taking into account the semantics given

in [15] through an extension of the asynchronous $\pi-$ calculus. The intuition is to associate at design time a scope with two processes: a compensation process and a fault handler.

The compensation process can be triggered only after the successful completion of the activity of the scope, in order to compensate the effects of the scope. The failure handler is triggered in case of failure during the execution of the scope and include (among other activities that depend on the specific process that is being modelled) the triggering of the compensations of all the scopes, enclosed in the failing scope, that have already terminated successfully.

The Scope operator is encoded in the following SRML transition:

```
transition X
    triggeredBy start[x] ∨ done[y]
    guardedBy ¬ policy[x]
    effects
    start[x] ⊃ ¬ start[x]' ∧ state[x]'=running ∧ start[y]'
    ∧ done[y] ⊃ ¬ done[y]' ∧ done[x]' ∧ state[x]'=exited
```

Transition *X* is executed twice:

1. when the triggering condition *start*[*x*] is enabled. The transition in this case has the following effects: (1) disabling the triggering condition *start*[*x*], (2) setting the state of task *x* to *running* and (3) enabling the triggering condition for the sub-task *y*.

2. when *y* terminates with success (i.e. *done*[*y*] = *true*). The transition in this case has the following effects: (1) disabling the successful termination of *y*, (2) enabling the signal of successful termination for *x* and (3) setting the state of *x* to *exited*.

5.4 Advanced Control Flow Encoding

In this section, we extend on the basic control flow encoding of StPowla notation operators by considering also the use of policy functions. We consider two types of policies: refinement policies and reconfiguration policies.

5.4.1 Refinement Policies

Recall that in StPowla refinement policies are those requesting the `req(action, [args], [SLR])` as action. The element that requires encoding in SRML is the Service Level Requirements list. This list contains expressions that specify acceptable values for a domain specific attribute: e.g. `[cup_temperature = warm]` which specifies that the chosen service (here a beverage service) should offer warm cups.

In SRML we have seen that the `EXTERNAL POLICY` element allows us to express such issues. The expression language in SRML is much more powerful than what StPowla offers currently: in SRML complex behaviour such as that seen in the earlier example can be expressed. The example did show a case where complete satisfaction was achieved by a delivery distance of less than 50 miles, and otherwise satisfaction did decrease in line with distance. In StPowla currently only simpler relations such as less, equal or more can be expressed but there will be ongoing work in enhancing the mechanisms in StPowla.

In general the mapping from StPowla to SRML involves creating an external policy for SLRs where for each StPowla attribute an `SLAVARIABLE` is created in SRML. The relation and values are then captured as a `CONSTRAINT` in the SRML policy rule.

Let us consider, for example, the following StPowla policy for the procurement service:

```
CheapService
appliesTo processOrder
    when on_task_entry
```

```
req(mail,[],[ServiceCost<10])
```

The policy *CheapService* ensures that the cost of the transaction with the service for processing the order is less then £10. In SRML we can express *CheapService* as a constraint that applies to the requires-interface OP of the module ProcurementService (see Figure 5.1). Since StPowla, at the moment, only allows to express sharp requirements, we use a constraint system where the degree of satisfaction has boolean values (i.e. {0, 1}).

```
EXTERNAL POLICY
_______________________________
SLA VARIABLES
        OP.SERVICECOST
CONSTRAINTS
        CheapService is <{OP.SERVICECOST},def1> s.t.
                if n<10 then def1(n)=1,
                otherwise def1(n)=0
```

The constraint uses the SLA variable OP.SERVICECOST (i.e. the price for enacting a transaction with the order processor), which assign degree of satisfaction of 1 if the service costs less then £10 and degree of satisfaction of 0 otherwise. The constraint ensures that any service which does not ensure a degree of satisfaction of 1 will not be selected.

5.4.2 Reconfiguration Policies

The key aim of this section is to illustrate how policies can influence the control flow and how this can be modelled in SRML. Thus we discuss the encoding of StPowla policies into SRML orchestrations. Each interaction is handled, in the orchestration of BP, by one or more transitions that model the policy handler. We will see in detail such transitions when discussing the single interactions, in the rest of this section.

A policy related to a task can have an effect (1) on the state prior to the task execution

(i.e. *delete*, *block* and *insert*) or (2) during the execution of a task (i.e. *fail* and *abort*). The state of a task is notified through the variable $state[x]$. The policy handler must check that the task is in the correct state according to the specific policy that has to be enacted. The policy handler prevents the execution of either (1) the task or (2) the rest of the task by using the variable $policy[x]$: the condition $\neg policy[x]$ guards the transition(s) corresponding to the execution of task. Notice that for most of the control constructs it is not possible to trigger policies of this second type on atomic tasks whose state changed directly from *toStart* to *done*.

Delete Task

The deletion of task (i.e. `delete(x)` in StPowla) skips the execution of x. The policy manager prevents the execution of x by signaling a policy exception (i.e. $policy[x] = true$).

```
transition policyHandler_delete_1
    triggeredBy delete[i]🔔
    guardedBy state[delete[i]🔔.task] = toStart
    effects policy[delete[i]🔔.task]'
```

Transition *policyHandler_delete_1* is triggered by the event $delete[i]$🔔, sent by PI. The guard ensures that a task can be deleted only if its execution has not started yet (i.e. its state has value *toStart*). The transition has the effect of setting the value of the variable *policy* to *true*. This will prevent the regular execution of the deleted task (recall that the execution of each task x is guarded by the condition $\neg policy[x]$). When the triggering condition for task x becomes *true*, the transition *policyHandler_delete_2* is executed instead of the transition of the deleted task.

```
transition policyHandler_delete_2
    triggeredBy start[x]
```

```
        guardedBy P_delete[i]🔔? ∧  delete[i]🔔.task=x
        effects ¬ start[x]' ∧ done[x]' ∧ state[x]' = done
        sends delete[i]✉
```

The guard of *policyHandler_delete_2* ensures its execution only if a deletion policy has been triggered for *x*. The effects of the transition are: (1) to disable the triggering condition of *x*, (2) to notify the correct termination of *x* (which in fact has not been executed) and (3) to set the state of *x* to *done*. The transition also sends the reply event to the interaction *delete* to notify PI of the completed enactment of the reconfiguration policy.

Block Task

The function block(x, p) in StPowla blocks a task until *p* is *true*. In SRML the policy handler prevents *x* from executing (i.e. *policy[x]* becomes *true*) temporarily until *p* is *true*. The policy handler notifies the enactment of the policy to the environment after that the task has been unblocked.

```
    transition policyHandler_block_1
        triggeredBy block[i]🔔
        guardedBy state[block[i]🔔.task] = toStart
        effects policy[block[i]🔔.task]'
```

Transition *policyHandler_block_1* is triggered by the event *block*[*i*]🔔, sent by PI. The guard ensures that a task can be deleted only if its execution has not started yet (i.e. its state has value *toStart*). The transition has the effect of setting the value of the variable *policy* to *true*. This will prevent the regular execution of the blocked task. When the condition specified through condition *p*, which has been communicated by PI, becomes true then the transition *policyHandler_block_2* unblocks the task.

```
transition policyHandler_block_2
    triggeredBy block[i]🔔.condition
    guardedBy P_block[i]🔔
    effects ¬ policy[block[i]🔔.task]'
    sends block[i]⌧
```

Transition *policyHandler_block_2* is triggered by the condition *block*[*i*]🔔*.condition* and the guard ensures its execution only the task has previously been blocked. The transition has the effect of setting the variable *policy* for the task to which the policy applied to *false* so that the task can be executed as soon as its triggering condition becomes true. The transition also sends the reply event to the interaction *block* to notify PI of the completed enactment of the reconfiguration policy.

Insert

The insertion of a task, represented by the function insert(y, x, z) in StPowla, inserts the task y in sequence or in parallel with respect to x depending by the value of the boolean variable z. In SRML the insertion is triggered by the interaction *insert*[*i*]⌧ with parameter *insert*[*i*]⌧*.task* representing the task x, *insert*[*i*]⌧*.insertedTask* representing the task y and *insert*[*i*]⌧*.condition* representing the condition z. We assume that the set of tasks for which it is possible to insert is determined a priori; in this way we assume that the SRML encoding has a set of transitions for each possible task, including the task to possibly insert, that is executed by setting *start*[y] to *true*. We introduce in this way a limitation on the number of task types that we can insert and on the fact that a task can be inserted only once (we will manage multiple insertions in the future, when we will encode looping constructs) but we do not provide any limitation on the position of the insertion.

We rely on a function $next : taskId \rightarrow taskId$ that returns, given a task, the next task to execute in the workflow. Such a function can be defined by induction on the syntax of StPowla defined in Chapter 4.

```
transition policyHandler_insert_1
    triggeredBy insert[i]🔔
    guardedBy  state[insert[i]🔔.task]=toStart
    effects policy[insert[i]🔔.task]'
```

The transition *policyHandler_insert_1* prevents the execution of the task on which the policy applies (i.e. *insert*[*i*]✉.*task*) by setting the its policy variable to true. When the task on which the policy applies is triggered, *policyHandler_insert_2* is executed instead of the regular transition for the task.

```
transition policyHandler_insert_2
    triggeredBy start[x]
    guardedBy P_insert[i]🔔 ∧ insert[i]🔔.task=x
    effects
    insert[i]🔔.condition ⊃ ¬ policy[insert[i]🔔.task]'
    ∧ ¬ insert[i]🔔.condition ⊃ policy[insert[i]🔔.task]'
                                ∧ start[insert[i]🔔.insertedTask]'
```

The transition *policyHandler_insert_2* starts the execution of the task on which the policy applies (in parallel with the inserted task if *insert*[*i*]✉.*condition* = *true*). The transitions *policyHandler_insert_sequence* and *policyHandler_insert_parallel* coordinate the execution of the tasks (the one on which the policy applies and the inserted one) in sequence or in parallel, according to the condition.

```
transition policyHandler_insert_sequence
    triggeredBy done[x] ∨ done[y]
    guardedBy P_insert[i]🔔 ∧ insert[i]🔔.condition
     ∧ (insert[i]🔔.task=x ∨ insert[i]🔔.insertedTask=y)
    effects done[x] ⊃ ¬ done[x]' ∧ start[y]'
    ∧ done[y] ⊃ ¬ done[y]' ∧ start[next(x)]'
```

```
        sends
        done[y] ⊃ insert[i]✉

    transition policyHandler_insert_parallel
        triggeredBy done[x] ∧ done[y]
        guardedBy P_insert[i]🔔? ∧ ¬ insert[i]🔔.condition
         ∧ insert[i]🔔.task=x ∧ insert[i]🔔.insertedTask=y
        effects ¬ done[x]' ∧ ¬ done[y]' ∧ start[next(block[i]🔔.task)]'
        sends insert[i]✉
```

Transitions *policyHandler_insert_sequence* and *policyHandler_insert_parallel* are similar to regular sequence and parallel transitions, but they are guarded by the fact that an insertion policy with positive/negative condition has been triggered in the past. The effects are similar to those of a regular sequence/parallel transitions. A reply event for the interaction *insert* is sent to notify PI of the completed enactment of the reconfiguration policy when, in the case of *policyHandler_insert_sequence* transition the inserted task terminates (i.e. *done*[*y*] ⊃= *true*) and in the case of *policyHandler_insert_parallel* both of the parallel tasks terminate.

Fail Task

The failure of a task must occur during the execution of the task (it has no effects otherwise). The failure can be triggered autonomously, within the task or induced externally by the execution of the policy *fail*. We consider here the second case.

```
    transition policyHandler_fail
        triggeredBy fail[i]🔔
        guardedBy state[fail[i]🔔.task]=running
        effects policy[i][fail[i]🔔.task]'
```

```
          ∧  state[fail[i]🔔.task]'=failed
    sends fail[i]✉
```

Transitions *policyHandler_fail* is triggered by the event *fail*[*i*]🔔, sent by PI. The guard ensures that a task can fail only if it is currently in execution (i.e. its state has value *running*). The transition has the effect of setting the value of the variable *policy* to *true* and the state of the task to *failed* (so that the normal flow of execution blocks). A reply event for the interaction *fail* is sent to notify PI of the completed enactment of the reconfiguration policy.

Abort Task

The abortion of a task is similar to a deletion, but it involves a running task. An abort of a task occurring not during its execution has no effects.

```
transition policyHandler_abort
    triggeredBy abort[i]🔔
    guardedBy state[abort[i]🔔.task]=running
    effects policy[i][abort[i]🔔.task]'
               ∧ state[abort[i]🔔.task]'=done
    sends abort[i]✉
```

Transitions *policyHandler_abort* is triggered by the event *abort*[*i*]🔔, sent by PI. The guard ensures that a task can fail only if it is currently in execution (i.e. its state has value *running*). The transition has the effect of setting the value of the variable *policy* to *true* and the state of the task to *done* (so that the flow of execution can continue normally). A reply event for the interaction *abort* is sent to notify PI of the completed enactment of the reconfiguration policy.

5.4.3 Reconfiguring the Procurement Scenario

The orchestration of the business role BusinessProtocol would consist of the sequence of the tasks request order (i.e. task *ro*) and process order (i.e. task *po*).

```
transition X
    triggeredBy start[x] ∨ done[ro] ∨ done[po]
    guardedBy policy[x]
    effects
    start[x] ⊃ ¬ start[x]' ∧ state[x]'=running ∧ start[ro]'
    ∧ done[ro] ⊃ ¬ done[ro]' ∧ start[po]'
    ∧ done[po] ⊃ ¬ done[po]' ∧ done[x]' ∧ state[x]'=exited
```

In case of a receive event of type $insert[i]⊠$, triggered by the component PI, with parameter *task* equal to *po*, parameter $insertedTask$ equal to *gbd* (i.e. get deposit), and the parameter *condition* equal to *true*, the policy handler would: (1) block the execution of *ro* (preventing in this way *ro* to trigger its continuation $po = next(ro)$) by setting $policy[ro] = policy[po] = true$, (2) wait for the condition $start[ro] = true$ that is triggered by transition X, (3) since the parameter *condition* is *true*, the policy handler would unblock *ro*, (4) the transition $policyHandler_insert_sequence$ would handle the execution of *gd* after *ro* and, finally, trigger *po* by setting $start[po] = true$.

5.5 Summary

In this chapter, we have expanded upon the relationship between StPowla and Service Oriented Architecture by providing an encoding from StPowla processes to SRML models. The latter is a modelling language for service oriented systems that provides a rich set of features for modelling almost every aspect of a service oriented application.

The reason for providing this encoding is to clearly define the business process of

StPowla with the orchestration model of services. SRML has clearly defined semantics and we have presented a methodology for the transposition from StPowla to SRML.

We presented two types of policy: refinement policies which govern how a process is carried out (including service selection constraints from the SRML perspective and execution constraints from the StPowla perspective) and reconfiguration policies which modify the execution of the workflow by substituting transitions for existing ones, following responses from the Policy Interface component.

In the next chapter, we will present a case study from from a real world situation and apply StPowla concepts to it. Furthermore, we will include the methodology of the encoding to SRML as defined in this chapter.

Chapter 6

Case Study

Many ideas grow better when transplanted into another mind than in the one where they sprang up. (Oliver Wendell Holmes).

6.1 Introduction

In the previous three chapters we have introduced StPowla, its reconfiguration functions and some basic encoding from StPowla to SRML. In this chapter, we apply these concepts to an industrial case study, provided by the Sensoria project. Firstly, we examine a given scenario from the telecommunications industry, with a core workflow and some points of variability. We express all information in StPowla terms and we examine the consequences of applying the variability rules. Secondly, we encode the given scenario with StPowla functionality into SRML, to demonstrate the relationship between StPowla and SRML and ultimately the relationship we see between workflows and Service Oriented Architecture.

6.2 Scenario

The case study is based on a Voice over IP (VoIP) procurement example between a customer and a reseller. If viewed at a higher level, it can represent most, if not all, procurement examples. Essentially, a customer makes an order for a VoIP service. The Order Management system that receives the order first performs some service-level tests and transmits the results to the customer. If the customer rejects the results, the process ends. If the customer accepts, the order management system prepares an offer and may optionally request legal assistance. The contract is then created.

The use case diagram is shown in Figure 6.1, using the notation for use-case diagrams proposed in [10]. The customized icons represent different types of roles that an actor can have in a service oriented context (e.g. dynamically discovered services, statically bound persistent resources, service requester, etc.).

The diagram models a service, provided to a service requester represented by the actor *Customer*, which manages the order for a VoIP connection. The service relies on the persistent resource *Legal*, which is shared among the different instances of the service, and the external service *Test* which is dynamically discovered and selected according to the customer's location.

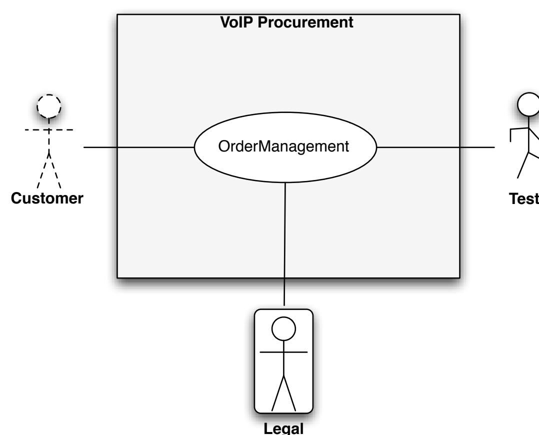


Figure 6.1: VoIP Case Study use-case diagram

The original workflow is shown in Figure 6.2. It contains four swimlanes, representing the four parties (or Actors) that participate in the workflow. Note that each swimline could be in itself a workflow for the individual actor, but for clarity the entire workflow is listed as one. For the purpose of this chapter, we will focus on the Order Management swimlane as it contains the key functionality we wish to examine. It is possible to consider solely this entity and ignore the Legal, Customer and Test entities, however the global view provides additional, useful context.

The general process is (1) an order is received, (2) tests are performed and the customer is requested to accept or reject the results, (3) if accepted, an offer proposal is created with assistance from a legal entity, (4) the contract is created.

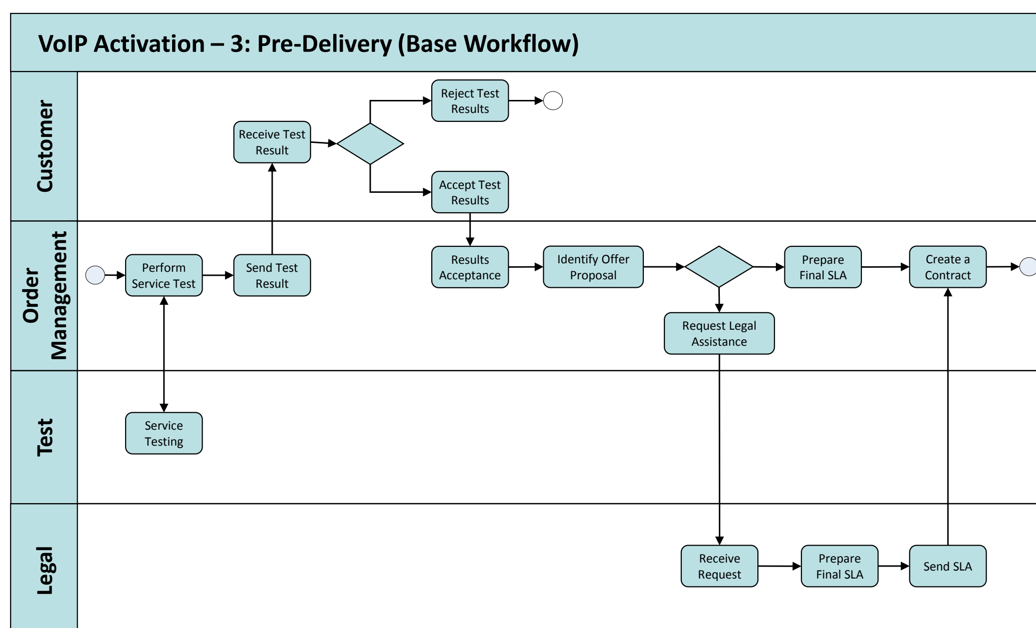


Figure 6.2: Case study workflow

There are two variants to this workflow:

1. If the customer is small and the order value is small, do not perform service tests.

2. If the order value is small, do not use legal assistance to generate the final contract.

From what we can see so far, the workflow given has at least two issues we must consider.

First, neither decision point (identified by the diamonds) have tests. In the case of the decision point in the Customer swimlane, we can assume that the decision (to accept or reject the test results) is the responsibility of the Customer actor. In the Order Management swimlane, again there is no condition and this time there is no clear assumption to be made. However, on reflecting on the second workflow variant, it seems most likely that the decision point is not necessary in the core workflow, and that the "Prepare Final SLA" task (in the Order Management swimlane) does not exist in the core workflow.

The second issue is that the task "Create A Contract" has two incoming control flows. If writing this in StPowla notation, a Flow Merge would be required before the task to merge the two control flows into a single flow prior to entering the task. However, since the above assumption renders one of the control flows non-existent, we can safely conclude this is no longer of any importance.

Following these assumptions, we refine the workflow to that shown in Figure 6.3. Notice particularly that the task "Prepare Final SLA" (in the Order Management swimlane) is inside the workflow model, but not connected to anything, i.e. it is currently redundant. We maintain it in the workflow to support the second workflow variant.

We now proceed to showing this case study in StPowla and the equivalent encoding in SRML.

6.3 StPowla Representation

For consistency, we translate the (refined) workflow into StPowla notation in Figure 6.4, whilst maintaining the swimlanes for familiarity.

Next we add the ontology as follows (tasks not listed for brevity):

```
Workflow BP is
    Invariants
```

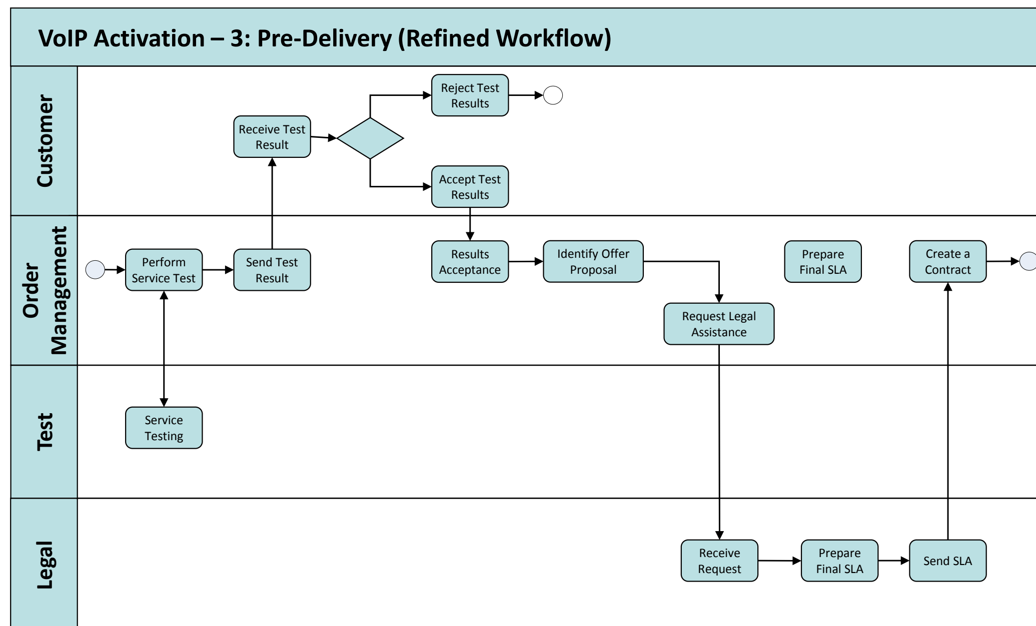


Figure 6.3: Refined case study workflow

```
    LowBusinessValueThreshold: int
    LowOrderValueThreshold: int
Actors/Entities
    Legal: Actor
    Test: Actor
    OrderManagement: Actor
    Customer: Actor
        Attributes
            int orderValue
            int businessValue
Tasks
    ...
Scopes
    s1: [OrderManagement.PerformServiceTest, ...,
            OrderManagement.ResultsAcceptance]
    s2: [OrderManagement.RequestLegalAssistance, ...,
            Legal.SendSLA]
```

In addition, Figure 6.5 shows the same workflow with the scopes marked out.

Importantly we note that the workflow has two invariants `LowBusinessValueThreshold` and `LowOrderValueThreshold`. These refer to the level of spend from a customer for which the customer is thus regarded as a small customer and the upper value of an order which would be considered as small,

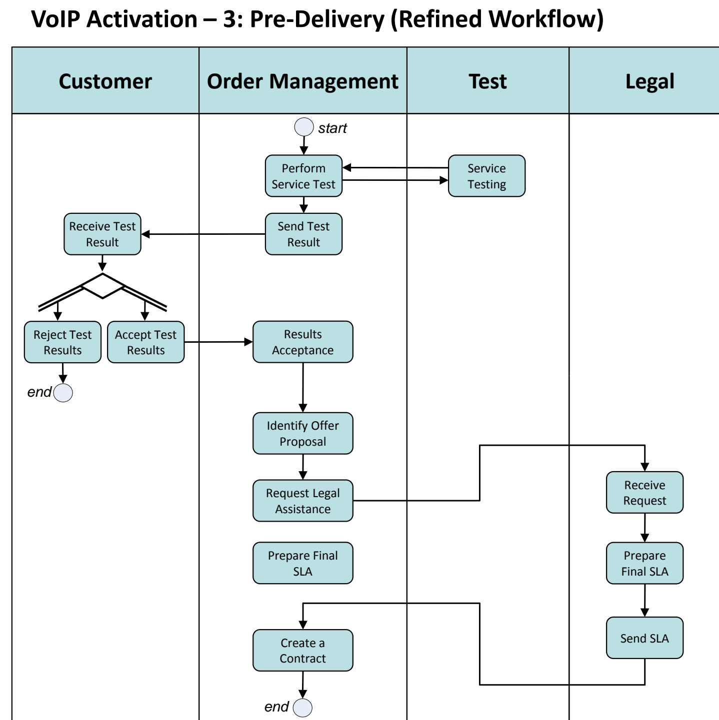


Figure 6.4: Case study StPowla workflow

respectively. How these values are ascertained is not important[1]. What is important is that the values are accessible.

Furthermore, on invocation of the workflow, two variables attached to the `Customer` actor specifying the customer's cumulative spend in the last year (`businessValue`) and the value of the current order (`orderValue`). Again, we are not concerned with how these values are obtained, more that we can reference them during workflow execution.

We translate the given variation rules to StPowla policies as follows:

```
policy P1 is
  appliesTo BP
       when s1.started
         if   Customer.businessValue < LowBusinessValueThreshold
            and
              Customer.orderValue < LowOrderValueThreshold
```

[1]The values could be obtained from an external task (e.g. a service) or passed as a parameter from the customer who invokes the workflow

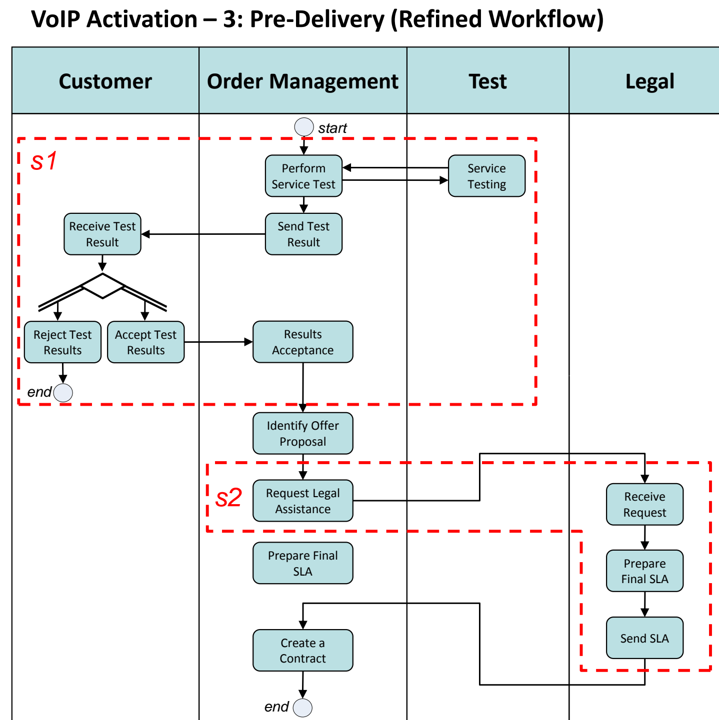


Figure 6.5: Case study StPowla workflow with scopes marked

```
        do delete(s1)


policy P2 is
  appliesTo BP
       when IdentifyOfferProposal.started
         if Customer.orderValue < LowOrderValueThreshold
         do   delete(s2)
            and
              insert(OrderManagement.PrepareFinalSLA,
                     OrderManagement.IdentifyOfferProposal,
                     false)
```

Policy $P1$ is straightforward, with the only complication that the use of the *Exceptional Delete* allowance is used. Its graph transformation rule is shown in Figure 6.6[2].

[2]For clarity, we will consider that in a single case both policies are applicable. Thus the graph transformation rules have been presented in a way that each can be applied to the output of the former rule, if one exists. Thus in this first rule, there are references to $s2$ and $PrepareFinalSLA$ which are not needed. Also, the workflow instance graph may be defined more than what is required for the rules.

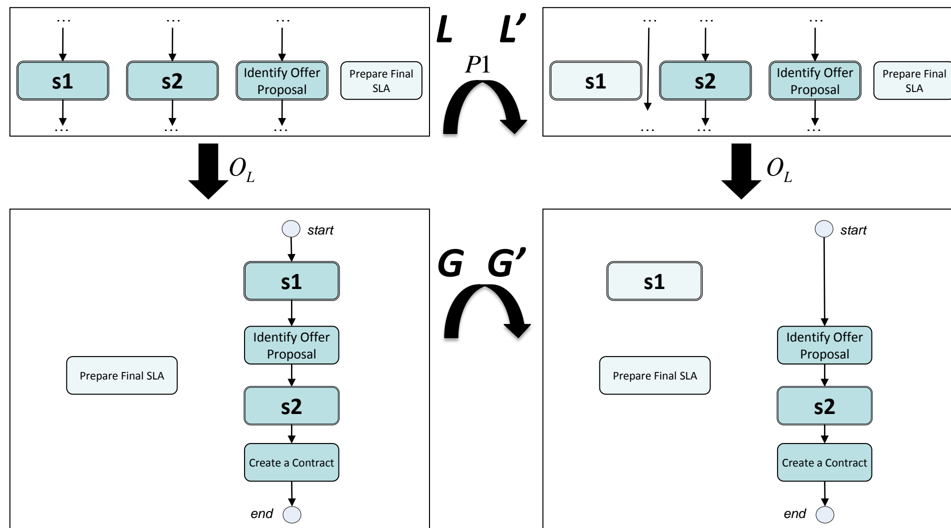


Figure 6.6: Policy $P1$ simply removes scope $s1$ from the workflow. In this diagram, this is depicted by the $s1$ task being removed from the workflow in both the type and instance graphs. It still remains available though for future reuse.

Policy $P2$ states that on commencing the *IdentifyOfferProposal* task, if the order value could be considered small, then the order management system should not invoke legal assistance and that they should prepare the final Service Level Agreement themselves (i.e. remove a scope and insert a task). This is thus a 2-step policy which we identify as step a and step b. Step a deletes the set of tasks that we do not need anymore. The graph transformation rule is shown in Figure 6.7. Step b inserts the new task into the workflow at the given position. Its graph transformation rule is shown in Figure 6.8.

Noting the trigger position of the policy and the initial area of effect, the trigger could also have been `workflow.started` or `IdentifyOfferProposal.task_completed` (for $P2$). The choice of which trigger to use is down to the author and this can have a significant effect depending on the existence of other policies. For example, if the trigger of the $P2$ was bound to an event from a task in scope `s1`, it is possible that the scope would be deleted and the policy never initiated. Whereas instead the required functionality was to initiate the policy regardless.

If the customer could firstly be identified as a small customer and also the order value is below the low order threshold, it is clear that both policies would be applied. As a result,

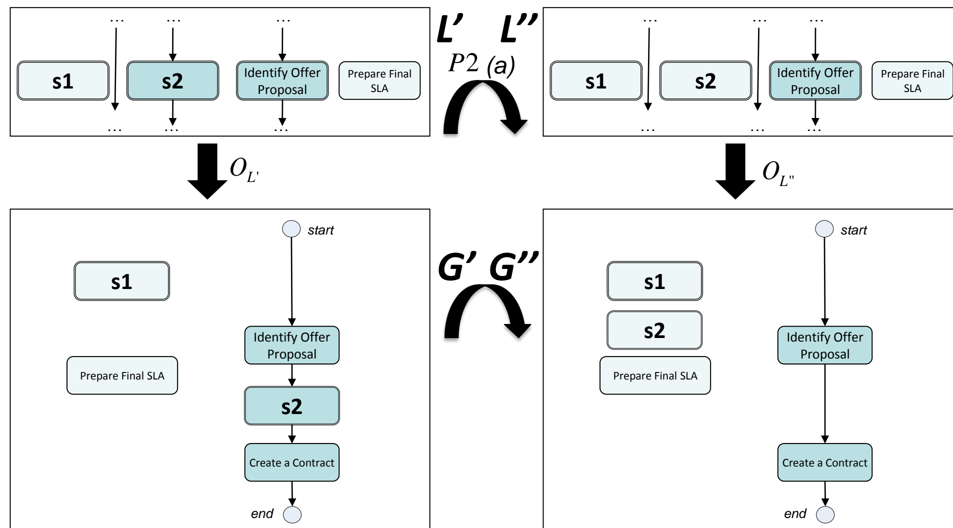


Figure 6.7: Policy $P2$ has two steps and therefore we provide graph transformation rules for step a and step b. In step a, scope $s2$ is removed from the workflow similarly to the policy $P1$.

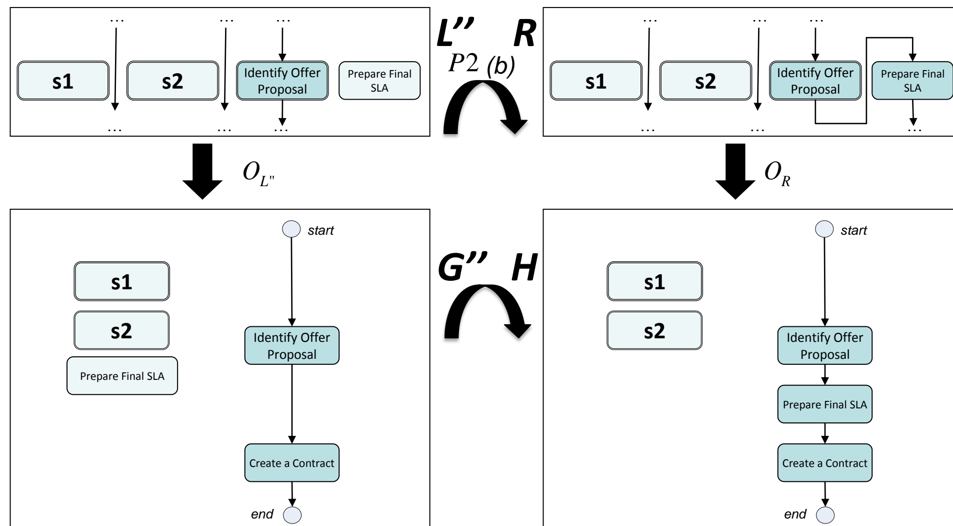


Figure 6.8: Step b inserts the *PrepareFinalSLA* task into the workflow after *IdentifyOffer-Proposal*.

both policies are applicable and thus both graph transformation rules are used.

In this section, we have taken the scenario and expressed the workflow and variants in StPowla terms, i.e. the workflow, ontology, policies and their respective graph transformation rules. In the next section, we encode the variants in SRML.

6.4 Encoding to SRML

In this section we encode the StPowla workflow and policies into SRML, following on from the presentation of StPowla concepts in Chapter 3, reconfiguration functions in Chapter 4 and their encoding to SRML in Chapter 5. We present a methodology for developing services, which involves StPowla and SRML. The discussion on the methodology further illustrates the rationale of the work presented in this thesis by discussing the benefits of a joint usage of SRML, StPowla and the encoding.

Some of the methodological aspects concerning SRML have been discussed in previous work [10], which proposes a process to arrive at (formal) service models in SRML starting from informal (or semi-formal) specifications in notations that are typically described in UML. In Figure 6.1 we use the extension of UML use case diagrams presented in [10] to capture requirements in a service oriented scenario and derive the structure of SRML models.

From the use case diagram we can determine the services and resources our application relies on. The definition of the internal structure of the SRML module (i.e. the components and wires that define the internal workflow) depends in general on the portfolio of components already available for reuse within the business organisation. The definition of a complex internal structure from scratch, deriving from the decomposition of the orchestration in a number of coordinated units, can be done, in general, using traditional techniques for Component Based Development.

SRML offers primitives based on events that allow us to suitably model those scenarios where process-based modelling could result in over-specification. However, the primitives of SRML are general enough to support process based modelling. For example, [14] presents an encoding from the process-base style of modelling of WS-BPEL to SRML. Another example is the encoding presented in this chapter where StPowla provides SRML with a means to define, at a higher level of abstraction, a process-based

behaviour (i.e. workflow schedule) which is dynamically reconfigured by policies.

6.4.1 Methodology

The methodology presented through the example involves three different languages that suit the needs of different stages of the modelling process:

Requirements UML Use Case diagrams are used to capture the requirements of the service oriented application that has to be developed. The application may involve a number of related services (i.e. they may share resources and are modelled as a single business unit). The usage of UML Use Case diagrams provides a human friendly notation suitable in this phase which involves the intervention of non experts in the engineering process (e.g. members of the business company commissioning the development of the application). The outcome of this phase is a Use Case diagram. By using the mapping defined in [10] it is possible to derive, from the diagram, the structure for a number of SRML modules where the orchestration and the behavioural specification of the external interfaces is left unspecified.

Business Modelling StPowla is used to perform the modelling, at the business level, of the orchestration in terms of workflow schedule and reconfiguration policies. This phase can be performed by experts in business modelling who can benefit by the level of abstraction provided by StPowla (i.e. modular definition of control-based process and business policies). The outcome of this phase is a StPowla model of the business process that, by using the encoding to SRML, can provide each of the SRML module structures resulting from the previous phase with an the internal orchestration.

Service Oriented Modelling The SRML modules obtained in the previous phase can be extended to include the behavioural specifications for the external interfaces, modified to include other components in the internal structure (for example component

extracted from implementations in BPEL [14] or any other language for which an encoding into SRML has been provided), and analyzed with the formal framework provided by SRML.

6.4.2 Use Case Driven Example

In this section, we present the encoding of the VoIP case study to SRML. The requirements are expressed in Figure 6.1, using the notation for use-case diagrams proposed in [10]. The customized icons represent different types of roles that an actor can have in a service oriented context (e.g. dynamically discovered services, statically bound persistent resources, service requester, etc.).

As mentioned in Section 5.2, SRML distinguishes between the different types of actors by representing nodes at different layers. The SRML module derived by the diagram in Figure 6.1 is illustrated in Figure 6.9[3]. The module has one provides-interface CR of type Customer, one bottom-layer interface LE of type Legal and one requires-interface TE of type Test. The internal structure of the module is defined according to Section 5.2: the component BP of type BusinessProcess defines the base workflows for orchestrating the service, the component PI of type PolicyInterface handles the reconfiguration requests.

According to the encoding, the process triggered by the root transition (see Section 5.3) is represented in SRML by the transition X which executes in sequence the scope $s1$, followed by task *IdentifyOfferProposal*, followed by scope $s2$ and finally task *CreateAContract*. For readability purposes, we identify the task *IdentifyOfferProposal* as $t1$ and task *CreateAContract* as $t2$.

```
transition X
```

[3] The clock symbol indicates that the related component has an internal configuration policy, which identifies the triggers of the external service discovery process as well as the initialisation and termination conditions of the components that instantiate the component-interfaces [30].

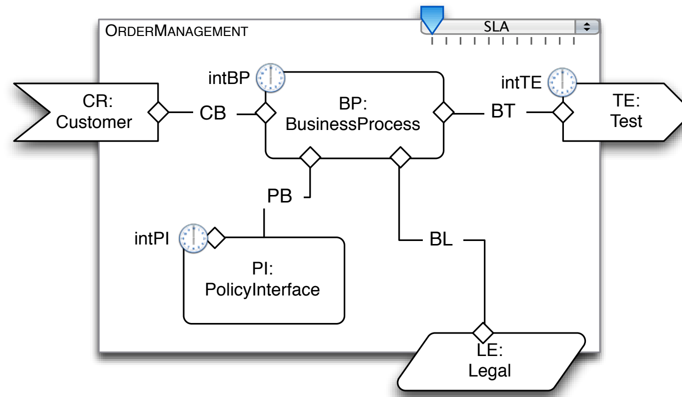


Figure 6.9: VoIP Case Study SRML module.

```
triggeredBy start[x] ∨ done[s1] ∨ done[t1] ∨ done[s2] ∨ done[t2]
   guardedBy ¬ policy[x]
   effects start[x] ⊃ ¬ start[x]' ∧ state[x]'=running ∧ start[s1]'
           ∧ done[s1] ⊃ ¬ done[s1]' ∧ start[t1]'
           ∧ done[t1] ⊃ ¬ done[t1]' ∧ start[s2]'
           ∧ done[s2] ⊃ ¬ done[s2]' ∧ start[t2]'
           ∧ done[t2] ⊃ ¬ done[t2]' ∧ done[x]' ∧ state[x]'=exited
```

Transition X[4] is executed when the sequence task is triggered and when any of the subtasks of the sequence terminate. The guard ensures that no policies have been triggered for x. The description of what happens in each execution is as follows:

First Execution: the state of x is set to running and $s1$ in the sequence is triggered.

Second Execution: $s1$ completed successfully and triggers $t1$.

Third Execution: $t1$ completed successfully and triggers $s2$.

Fourth Execution: $s2$ completed successfully and triggers $t2$.

[4]The careful reader will have noticed that transition X contains more than two tasks to be executed in sequence. Whereas in the previous chapter we only identified a sequence transition as having two tasks, we can easily include more as shown. For our purposes, it is far easier and clearer to define one transition containing many tasks in sequence, rather than a set of transitions that coordinate themselves and eventually two tasks. In this case study, transitions would be created to go from $s1$ to $t1$, then from $t1$ to $s2$, then from $s2$ to $t2$ then finally the completion of X.

Fifth Execution: $t2$ completed successfully and sequence task x terminates successfully.

Fragments of the transitions scope1 and scope2, which illustrate triggers and guards, are reported below. The variables y and z represent the root process started within each scope.

```
transition scope1
    triggeredBy start[s1] ∨ done[y]
    guardedBy ¬ policy[s1]
     ...

transition scope2
    triggeredBy start[s2] ∨ done[z]
    guardedBy ¬ policy[s2]
     ...
```

The business role BusinessProcess provides transitions to handle the reconfiguration for each sub-process. When the customer creates a new instance of the service OrderManagment, the SLA variables concerning the business value of the customer and the order value are set. Depending on those values, PI will either request to apply $P1$ and $P2$, or not.

In order to apply $P1$, for instance, PI sends an event $delete[s1]$🔔! to BP. The transitions policyHandler_p1_1 and policyHandler_p1_2, defined as described in Section 5.4, handle the request of the component PI to delete the scope $s1$ (i.e. $P1$).

```
transition policyHandler_p1_1
  triggeredBy delete[s1]🔔
     guardedBy state[delete[s1]🔔.task] = toStart
     effects policy[delete[s1]🔔.task]'

transition policyHandler_p1_2
```

```
   triggeredBy start[x]
      guardedBy P_delete[s1]🔔 ∧ delete[s1]🔔.task=s1
      effects ¬ start[s1]' ∧ done[s1]' ∧ state[s1]' = done
      sends delete[s1]✉!
```

The transition policyHandler_p1_1 is triggered by the deletion request, only if scope $s1$ has not started its execution yet, and sets the variable *policy*[*delete*[$s1$]🔔.*task* to true. When the transition x sets $start[s1] = true$, the transition s1 cannot be triggered because of the false guard. The transition policyHandler_p1_2 is triggered instead.

The effects of policyHandler_p1_2 are to set the variables for $s1$ as if the scope had been successfully executed (but in fact it has not been executed at all), and to notify PI of the deletion through the interaction *delete*[$s1$]✉!. The next transition to be executed is, again, x which triggers the next process in the sequence (i.e. $t1$). After $t1$ has completed, $s2$ would be executed under normal circumstances. However, if $P2$ was applicable, BP is subject to two further reconfigurations.

For step a of policy $P2$, the transitions are similar for deleting scope $s2$:

```
transition policyHandler_p2_1_1
   triggeredBy delete[s2]🔔
      guardedBy state[delete[s2]🔔.task] = toStart
      effects policy[delete[s2]🔔.task]'

transition policyHandler_p2_1_2
   triggeredBy start[x]
      guardedBy P_delete[s2]🔔 ∧ delete[s2]🔔.task=s2
      effects ¬ start[s2]' ∧ done[s2]' ∧ state[s2]' = done
      sends delete[s2]✉
```

These transitions are essentially the same as those for $P1$, therefore we do not describe

them here. However, after these transitions have been executed, we must also consider the second part of $P2$, i.e. inserting $t2$.

For step b of policy $P2$, the transition performs an insert as follows:

```
transition policyHandler_p2_2_1
   triggeredBy insert[t1]🔔
      guardedBy state[insert[t1]🔔.task] = toStart
      effects policy[insert[t1]🔔.task]'
```

This transition is from the Policy Interface and is triggered when the insert policy is enabled. The effects are guarded by the existing task's state, which should not have started yet. The effects are to modify the original transition's policy guard to being true.

```
transition policyHandler_p2_2_2
   triggeredBy start[x]
      guardedBy P_insert[t1]🔔 ∧ insert[t1]🔔.task=t1
      effects insert[t1]🔔 ⊃ ¬ policy[insert[t1]🔔.task]'
              ∧ ¬ insert[t1]🔔.condition ⊃ policy[insert[t1]🔔.task]'
                                  ∧ start[insert[t1]🔔.insertedTask]'
      sends insert[t1]✉
```

Transition policyHandler_p2_2_2 replaces the original and starts the execution of $t1$ as this is the task that the policy applies to. The condition in this case is false (for inserting in sequence) and the effects are to send the policy signal for inserting the new task and to send the start signal of the inserted task.

```
transition policyHandler_p2_insert
   triggeredBy done[t1] ∨ done[t2]
   guardedBy P_insert[t1]🔔 ∧ insert[t1].condition
            ∧(insert[t1].task=t1 ∨ insert[t1]🔔.insertedTask=t2
```

```
effects done[t1] ⊃ ¬ done[t1]' ∧ start[t2]'
        ∧ done[t2] ⊃ ¬ done[t2] ∧ start[next(t1)]'
sends done[t2] ⊃ insert[t1]🔔
```

This final transition coordinates the execution of the tasks. When the done signal $t1$ is received, this signal is disabled and the start signal of $t2$ is sent. When this is received (i.e. $t2$ completes), the signal is disabled and the next task after $t1$ originally is sent its start signal (through the use of the `next()` function.).

Thus these are the SRML transitions applicable to this case study.

6.5 Summary

In this chapter, we have taken a case study provided by an industrial partner on the SENSORIA project. The case study described a workflow based on VoIP provision, but can be seen at a higher level as representative of most, if not all procurement situations. Having first identified and resolved two design issues, we expressed both the workflow and its two (informally) written variants in STPOWLA (workflow notation and policies). We then applied the policies to the workflow through the use of the graph transformation rules defined in the previous chapter. Finally, we encoded the STPOWLA representation into SRML.

As a result, we have shown how STPOWLA can be applied to real situations. We have ascertained that STPOWLA can be mapped easily to such situations and enables workflow variations to be expressed in a manner that eliminates the use of natural English for definition and thus removes ambiguities. It also facilitates the design and implementation of a STPOWLA workflow engine that can automate the workflow execution process specifically in terms of applying variability as would be expressed in policies.

In the next chapter, we will evaluate the STPOWLA approach using a number of different measures, in order to determine adequacy, effectiveness and business value.

Chapter 7

Evaluation

7.1 Introduction

The main aim of this chapter is to take an objective view of what we have presented, namely the StPowla combination of workflows, policies and Service Oriented Architecture, but with a specific focus on reconfiguration functions that can be used in policies to affect the structure of the workflow. We will discuss the capabilities and limitations of our approach and evaluate the adequacy of the reconfiguration functions using workflow patterns as the benchmark. We will briefly discuss the area of policy conflict before concluding with an evaluation of the business value of our approach.

7.2 Capabilities and Limitations

Our original aim was to propose a solution to dynamic workflows, where SOA provided the underlying functionality. Change was to be made at the business level, rather than the orchestration level.

Through StPowla, we have defined an adequate means for defining workflows using a vi-

sual modelling technique. Although the number of operators available is relatively small, we consider this to be greater than a minimal set. For example, Strict Preference can be defined as a composition of tasks together with complex routing of outgoing control flow paths from the tasks. Although Random Choice is similar to this, its semantics are sufficiently different to warrant its own operator.

The reconfiguration functions represent a set of graph transformation functions applicable to StPowla workflows. We have defined the capability to insert tasks anywhere inside the workflow, either in sequence to other tasks or in parallel with them. The delete function offers the capability to avoid performing a particular function. These two functions alone provide scope for wholesale changes to the workflow.

Supporting those two essential functions are block, fail and abort. These three functions have the capability of changing the workflow execution without modifying the workflow structure. Whilst we call them reconfiguration functions, one might argue that they are not reconfiguration functions but execution functions.

So far, we are unable to define (in policies) completely new tasks. This is partially due the fact that we have not focussed on expressing task requirements in this thesis. The aim would be to have policy syntax that can be used for precisely this reason and thus remove the need for reconfiguring workflows with respect to only those tasks that the designer knows about. Once this capability has been introduced, it would be possible to have all workflows starting execution with no tasks, and from there new tasks are added through policies. This was the first of our two extreme situations which we recall from Chapter 1: Workflow *A* specifies no tasks at all and the control flow is empty. It is populated with tasks through policy functions at runtime. Workflow *B* is a significantly complex workflow with every single possible condition identified and embedded into the control flow. Neither of these extremes are adequate solutions and we are looking for a suitable balance between the two, which StPowla provides.

A second key limitation is that there is no existing capability for cancellation and com-

pensation functions. We have provided functions for aborting and failing tasks, but the semantics of these functions simply directs the control flow output to another branch (if one exists). There could potentially be the requirement to fail or abort a task and roll back its functionality to a given point, if such functionality exists. For example, suppose a person was at home cooking dinner when their partner returned home from work with pre-cooked food from the local takeaway. In this situation, the cooking would be aborted and some form of rollback would start, including returning uncooked food to the fridge and pantry, and disposing of any unusable food. Alternatively, the cooking may be paused whilst dinner is eaten, then completed and the food stored. In the former situation, we can see a compensation function in use, whereas the latter is more of a reconfiguration.

In the next section, we will examine StPowla further, including its notation and reconfiguration functions, against workflow patterns.

7.3 Workflow Patterns

A pattern "*is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts*" [76]. Workflow patterns are thus a set of frequently-recurring structures within workflows, including control flows, data flows and exception flows, to name only a few[1]. In this thesis, we restrict ourselves to considering only workflow patterns on control flows for obvious reason.

Workflow patterns have gained substantial exposure since their first publication in [93], followed by a revision in [78] and advanced patterns defined in [89]. We use them as a benchmark to evaluate StPowla against as they are well understood, widely accepted and comprehensively documented.

For each workflow pattern, we proceed to give a brief description, a discussion of how it is currently supported in StPowla, if it is also supported through reconfiguration functions

[1] See http://www.workflowpatterns.com for comprehensive documentation about workflow patterns

and any identified issues. Table 7.1 provides an overview ('✓' indicates full support for the workflow pattern, '-' indicates partial support and '✗' indicates no support).

7.4 Critical Assessment

The purpose of a comparison with workflow patterns is not simply to test the strength of capability, i.e. to see what possible workflow patterns can be supported, but more importantly to see how the StPowla approach facilitates the expression of different workflow patterns, and whether policies can play an integral part in providing such support. Simple patterns can of course be supported in several notations, including the one we have used in this thesis. But the evaluation of those patterns means little without having in mind the key element of StPowla, that is the separation of the core process to variability.

Thus, our evaluation is designed, in the first part, to see not just how many workflow patterns are supported in StPowla, but to evaluate the potential impact of policies and, more importantly, reconfiguration functions on a workflow. As will be seen in the following descriptions, the quantity of workflow patterns supported is the majority. The commentaries on each will often point to how policies are used to assist in the support of those patterns.

So our evaluation extends not just to determining capability of StPowla, but crucially the role that reconfiguration functions can play in the provision of such capability. Indeed, we will find that many patterns require reconfiguration functions to modify the workflow in a particular way. In fact, one might also consider that reconfiguration functions permit the dynamic creation of patterns at runtime, but to cover such a topic would require significant more space than is available here. What we learn is that the use of policies significant extends the capability of a workflow to implement a number of different situations. A simple workflow notation is only capable of so much, but the ability to modify a given model to an almost unlimited extend gives real expressive power to StPowla.

Table 7.1: Workflow patterns evaluation summary (preceding numbers refer to the workflow pattern ID at `http://www.workflowpatterns.com`)

Pattern	Supported
Basic Patterns	
1. Sequence	✓
2. Parallel Split	✓
3. Synchronisation	✓
4. Exclusive Choice	✓
5. Flow Merge	✓
Advanced and Synchronisation Patterns	
6. Multi-Choice	✓
7. Structured Synchronising Merge	✓
8. Multi-Merge	✓
9. Structured Discriminator	✓
28. Blocking Discriminator	✓
29. Cancelling Discriminator	✓
30. Structured Partial Join	✓
31. Blocking Partial Join	✓
32. Cancelling Partial Join	✓
33. Generalised AND-Join	✓
37. Local Synchronising Merge	×
38. General Synchronising Merge	×
41. Thread Merge	-
42. Thread Split	-
Multiple Instance Patterns	

Continued on Next Page…

Table 7.1– Continued

12. Multiple Instances without Synchronisation	×
13. Multiple Instances with a Priori Design-Time Knowledge	×
14. Multiple Instances with a Priori Run-Time Knowledge	×
15. Multiple Instances without a Priori Run-Time Knowledge	×
34. Static Partial Join for Multiple Instances	×
35. Cancelling Partial Join for Multiple Instances	×
36. Dynamic Partial Join for Multiple Instances	×
State-based Patterns	
16. Deferred Choice	✓
17. Interleaved Parallel Routing	-
18. Milestone	✓
39. Critical Section	-
40. Interleaved Routing	×
Cancellation and Force Completion Patterns	
19. Cancel Task	✓
20. Cancel Case	✓
25. Cancel Region	-
26. Cancel Multiple Instance Activity	×
27. Complete Multiple Instance Activity	-
Iteration Patterns	
10. Arbitrary Cycles	×
21. Structured Loop	-
22. Recursion	×
Termination Patterns	
11. Implicit Termination	×
43. Explicit Termination	-

Continued on Next Page. . .

Table 7.1– Continued

Trigger Patterns	
23. Transient Trigger	-
24. Persistent Trigger	-

We have compared StPowla notation and the reconfiguration functions to the widely-researched and documented workflow patterns, evaluating StPowla against 43 workflow patterns, with results showing 19 of those patterns are supported directly either through the normal notation or through reconfiguration functions (in some cases reconfiguration functions are essential to supporting the workflow patterns). Furthermore, StPowla provides indirect support for another 10 patterns. Of the remaining 14 patterns, 7 are to do with multiple instances, which StPowla does not support. The remaining 7 patterns relate to iterations, termination, synchronisations based on runtime information and the ordering of task executions at runtime.

The following sections provide further details on the evaluations.

7.4.1 Basic Patterns

Sequence:

Description:	An activity is enabled after the previous has completed.
Support:	Yes - this is supported directly in StPowla through the sequence operator.
Discussion:	This can be supported both through the basic notation and through reconfiguration functions.

Parallel Split:

Description:	Divergence of one branch into two or more parallel branches which execute concurrently.
Support:	Yes - this is directly supported in StPowla through the Flow Split operator.
Discussion:	This can be supported both through the basic notation and through reconfiguration functions.

Synchronisation:

Description:	Converges two or more branches into a single subsequent branch once all input branches have been enabled.
Support:	Yes - this is directly supported in StPowla through the Conditional Merge operator.
Discussion:	StPowla requires a Conditional Merge construct to be used in conjunction with a Flow Split in order to maintain structural integrity of the workflow (i.e. one control flow in and one control flow out). Thus, both the basic notation and reconfiguration functions will support this pattern. The purpose of forcing the combination of Flow Split and Conditional Merge operators was given for the pragmatic reason that it would like lead to less issues in workflow design, whereas multiple active control flows could potentially become available, and be hard to keep track of, if a Flow Split could be used in isolation from the Conditional Merge.

Exclusive Choice:

Description:	Divergence of control flow from one activity to one of a set of outgoing branches, with the path determined by the outcome of a logical expression.
Support:	Yes - this is directly supported in StPowla through the Flow Junction construct.
Discussion:	StPowla permits this "binary choice" pattern through both basic notation and through reconfiguration functions, in association with a Flow Merge operator. The concept of the Exceptional Delete was introduced partially to deal with situations in which multiple choice operators, such as this, lead to multiple merges which could make the workflow potentially unreadable. Thus, when an output from a choice operator leads to the end of the workflow, an early termination operator is allowed, but only as a shorthand method of writing in the Flow Merge operators.

Flow Merge:

Description:	Merges two or more branches without synchronisation.
Support:	Yes - this is directly supported in StPowla through the Flow Merge construct.
Discussion:	This pattern is only supported in conjunction with a flow divergence point, namely the Flow Junction, Strict Preference and Random Choice operators. It is supported both in standard notation and in reconfiguration functions.

7.4.2 Advanced Branching and Synchronisation Patterns

Multi Choice:

Description:	A variant of Exclusive Choice in which the conditions for each output branch are not disjoint.
Support:	Partial - StPowla provides the Flow Junction concept as a binary operator. The use of multiple Flow Junctions, specified with the correct condition checks, can model this pattern, albeit in more verbose terms.
Discussion:	This pattern is partially supported for the given reasons by both the standard notation and reconfiguration functions. It could be helpful to permit the user to change the condition through a reconfiguration function, however that could potentially lead to confusion at runtime (i.e. remembering which operator has which condition), with stacked Flow Junction conditions possibly contradicting each other (e.g. "*if (x < y) then if (x > y) then do something*").

Structured Synchronising Merge:

Description:	The convergence of two or more branches (that were previously diverged) into a single subsequent branch. It accompanies the Multi-Choice or OR-Split constructs.
Support:	Partial - as per Multi-Choice pattern.
Discussion:	See the discussion on the Multi-Choice pattern.

Multi-Merge:

Description:	Converges two or more branches into a single subsequent branch.
Support:	Yes - this is supported through the Flow Merge operator.
Discussion:	StPowla supports this pattern directly in the notation and reconfiguration functions. It can only be used in conjunction with a Flow Junction, Strict Preference or Random Choice operator to ensure workflow integrity. It could be useful to have one Flow Merge operator for multiple control flow divergence operators, although this is simply a shorthand way of writing multiple Flow Merge operators.

Structured Discriminator:

Description:	Converges two or more branches into a single subsequent branch following a previous divergence in the process. Also, control is passed to the subsequent branch when the first incoming branch is enabled.
Support:	Yes - this is supported through configuring the Conditional Merge operator.
Discussion:	StPowla provides support for this pattern directly in the Flow Split/Conditional Merge operators. The latter is configured using design time knowledge to determine the number of branches that must complete and also which of the branches are mandatory. Reconfiguration functions do not easily support modifiable Conditional Merge operators since new operators are typically either already defined or have preset values (i.e. 2 mandatory branches and both incoming branches must complete). It could be useful to extend this to allow reconfiguration of the Conditional Merge operator itself to allow runtime changes based on reactions to events. However, it can be safely assumed that almost any event can lead to a policy enforcing changes to the workflow which would be cleaner than simply changing values of an operator.

Blocking Discriminator:

Description:	The convergence of two or more branches into a single subsequent branch following one or more previous divergences. Control is passed to the subsequent activity when the first branch completes. Enablements can be blocked until the discriminator has been reset.
Support:	Partial - StPowla provides support through the configurable Conditional Merge operator.
Discussion:	The use of multiple divergence operators into a single convergence operator can be viewed as shorthand, or even sloppy modelling. StPowla requires 1-1 mapping and this can still model this pattern. Arguably our approach requires more design time work, although the introduction of the capability to have a single Conditional Merge operator for multiple Flow Split operators would just be a shorthand. A slight constraint in StPowla is that the operator is generally executed only once, or at least in isolation from other execution times. Thus no state information is kept by the operator between executions.

Cancelling Discriminator:

Description:	As per the Blocking Discriminator with the exception that on passing control to the subsequent activity, all remaining activities are cancelled.
Support:	Partial - this is not supported by StPowla notation, but reconfiguration functions can be used to abort or delete any remaining tasks that are no longer required.
Discussion:	Since the remaining tasks do not have a link to the active control flow, their current flow will either expire (i.e. come to a natural end) when the execution has completed or will undergo the abort path, executing any task (unless already deleted) and then expiring. The workflow/policy designer must take special care to avoid executing tasks on the abort path if they are not required.

Structured Partial Join:

Description:	Following a prior divergence, control is passed to a single subsequent branch when a specific number of incoming branches have been enabled.
Support:	Yes - this is supported through the configuration of the requirement of incoming branches in the Conditional Merge operator.
Discussion:	The support provided by StPowla requires the designer to know at design time how many branches are required to complete. This is a constraint on the general workflow pattern, which would add some potentially valuable functionality. Potentially, a policy function could be introduced to manipulate the configuration settings of operators.

Blocking Partial Join:

Description:	As per the Blocking Discriminator with the exception that a specific number of the incoming branches must complete.
Support:	Yes - through the Conditional Merge operator.
Discussion:	The Conditional Merge operator should be configured at design time as part of a sub-workflow scope in order to take advantage of configuring the number of branches required to complete.

Cancelling Partial Join:

Description:	As per the Cancelling Discriminator combined with the Blocking Partial Join. After the specific number of incoming branches have completed, the remainder are cancelled as control is passed to the single subsequent activity.
Support:	Yes - as per the Cancelling Discriminator and the Blocking Partial Join, i.e. delete policies should be used.
Discussion:	As per the Cancelling Discriminator and Blocking Partial Join.

Generalized AND-Join:

Description:	Convergence of two or more branches into a single subsequent branch after all incoming branches have completed.
Support:	Yes - through the Conditional Merge operator.
Discussion:	As per the Blocking Partial Join.

Local Synchronising Merge (also known as Acyclic Sychronising Merge):

Description:	The convergence of two or more branches into a single subsequent branch where the incoming branches were previously diverged. The number of branches that require synchronisation is determined by information in the workflow, including information given by the divergence construct or local data arriving at the merge.
Support:	No - StPowla supports only the configurable Conditional Merge.
Discussion:	It could be a valuable addition to StPowla to allow policy functions to modify the configuration information of each Conditional Merge. Furthermore, such enhancements could be made to other operators (e.g. modifying the condition check of Flow Junctions). It could however be advantageous to allow the workflow definition to reuse dynamic information, e.g. in condition checks, the variables addressed could be done so by reference rather than value. This means that no static values need to be defined at all.

General Synchronising Merge:

Description:	As per the Local Synchronising Merge with the exception that control is passed when either each incoming branch is enabled or will never achieve the enabled state.
Support:	No - as per the Local Synchronising Merge.
Discussion:	See the comments on the Local Synchronising Merge. StPowla does not support this pattern as it gathers information from the workflow directly whereas in StPowla the information must be known at design time.

Thread Merge:

Description:	A nominated number of execution threads are converged into a single branch of the same process instance.
Support:	Yes - StPowla provides support using the Flow Split and Conditional Merge operator.
Discussion:	Whereas the workflow pattern permits this construct to be defined “cleanly” (i.e. with minimal notation), StPowla requires additional notation to model this pattern. Thus there is no evidence to suggest a specific new operator for StPowla is required.

Thread Split:

Description:	A single execution branch can initiate a nominated number of execution threads.
Support:	Yes - StPowla provides support using the Flow Split and Conditional Merge operator.
Discussion:	See the Thread Merge pattern.

7.4.3 Multiple Instance Patterns

Currently StPowla does not support multiple instance patterns. Thus, no reconfiguration functions support multiple instances either. The reason for not supporting multiple instances is that a single task can be itself defined to invoke a service multiple times, i.e. the multiple instance behaviour can be embedded into the task itself. Also, through the use of a loop construct (a combination of Flow Junction, Flow Merge and an ontology variable), an alternative approach of using a structured loop can be used instead to similar effect.

7.4.4 State-based Patterns

Deferred Choice:

Description:	A set of branches represent possible execution paths at a single point. The decision is made by invoking the first activity of each branch and selecting the subsequent output branch from the first task to complete successfully.
Support:	Yes - this is supported through the Random Choice operator.
Discussion:	StPowla currently supports this pattern through the notation and through reconfiguration functions, noting that the Random Choice operator is used in conjunction with a Flow Merge operator.

Interleaved Parallel Routing:

Description:	A set of activities has a partial ordering that is finally determined at runtime, with each activity being executed only once and not concurrently with another.
Support:	Partial - StPowla supports full ordering or parallel processing.
Discussion:	Currently StPowla requires that the workflow designer explicitly state the execution order of all items. The only exception is that a set of tasks could be executed in parallel. The implementation ultimately determines the order in which the tasks are executed, however to the designer, the only requirement is that each task is completed prior to the subsequent activity commencing. Such a solution is also supported through reconfiguration functions.

Milestone:

Description:	An activity is only enabled when it is in a specific state.
Support:	Yes - this is supported through each tasks' *ready* variable.
Discussion:	StPowla supports this pattern both through the notation and through policies. However, the usage of this pattern does not constitute a reconfiguration since the workflow structure remains the same as before. A policy must still be used through to set the value of the *ready* variable of the target task.

Critical Section:

Description:	Two or more connected subgraphs of a process model are identified as "critical sections" and activities in each are executed in a mutually-exclusive way with other critical sections, i.e. a critical section must complete before another critical section may begin.
Support:	Partial - this is not supported directly in StPowla notation, but reconfiguration functions can emulate the pattern.
Discussion:	StPowla policies can be used to change tasks' *ready* attributes in order to control when they can and cannot start. Although not directly modelling the critical section pattern, this method does at least provide some means for substantial resemblance. To aid understanding, a critical section could be defined as a composite task and a policy could be created on one task that disables any others on entry, and enables them on completion.

Interleaved Routing:

Description:	Each activity in a set must be executed once, but there is no restriction on order except that none may be executed concurrently with another.
Support:	No - StPowla requires total ordering of tasks.
Discussion:	This pattern may be seen more as an implementation, or run-time constraint. Consider 10 people who wish to enter a garden through a gate. The order in which they enter does not matter, but the gate has the capacity for only 1 person at a time. The workflow designer is likely to specify the former aspects to this problem (*what*), without focussing *how*, which is the role of Services. Thus, there is insufficient technical substance for including this pattern in StPowla.

7.4.5 Cancellation and Force Completion Actions

Cancel Activity:

Description:	An enabled activity is withdrawn before commencing execution.
Support:	Yes - this is synonymous with using the Delete reconfiguration function on a task.
Discussion:	The StPowla approach is to remove the target task from the workflow, although not necessarily deleting its definition (i.e. it can be reused again later).

Cancel Case:

Description:	Removal of a complete process instance, including those activities currently being executed.
Support:	Partial - this can be emulated by deleting all remaining tasks and aborting the current task.
Discussion:	The main difference with the StPowla implementation is that any specific functionality defined for the *abort* event of the subject task will be executed, unless a policy changes that beforehand. Thus, we can argue that the StPowla approach provides greater flexibility, in contrast with a little extra work from the designer.

Cancel Region:

Description:	This involves the ability to cancel a set of activities that may or may not be currently executing within a process instance. Neither are the activities required to be connected to the overall process model.
Support:	No - StPowla places a restriction that any task must be connected to the main control flow, otherwise it will not be executed. The closest match to a region is a scope.
Discussion:	A useful implementation could be that a defined scope can be aborted and then the scope has its own abort control flow output defined. Alternatively, it would be straightforward to say that if a scope is failed, the active task within the scope is aborted and its abort control flow output is invoked. This would provide a broader range of functionality than just this workflow pattern.

Cancel Multiple Instance Activity:

Description:	This involves the cancellation of all instances (the number of which is known at design time) that are running concurrently and independent of each other.
Support:	No - StPowla does not support multiple instances of tasks.
Discussion:	n/a

Complete Multiple Instance Activity:

Description:	This involves the (sometimes forceful) completion of all instances of an executing activity.
Support:	Partial - StPowla does not support this pattern
Discussion:	Neither the StPowla notation or reconfiguration functions provide support for this pattern. The only possible ways for a task to complete is if it completes successfully, if it fails or if it aborts. To allow a task to complete "successfully", even when it failed or was aborted is counter-intuitive to design. However, it can be argued that some tasks are designed never to complete (e.g. calculating the value of π) and a completion event would put a safe end to the current processing. It can be seen though, that this is a slightly different interpretation of the abort functionality, from which support can be provided through a StPowla function.

7.4.6 Iteration Patterns

Arbitrary Cycles:

Description:	Cycles which have more than one entry/exit point.
Support:	No - StPowla does not support arbitary cycles.
Discussion:	The basic design of a StPowla workflow assumes one control flow input and (presuming a divergence operator is paired with the appropriate convergence operator) one control flow output, i.e. one entry and one exit point. To introduce multiple entry and exit points would potentially lose integrity and importantly lose the capability to track the workflow.

Structured Loop:

Description:	The ability to execute an activity or sub-process repeatedly.
Support:	Partial - no explicit support, however through the definition of a Flow Junction/Flow Merge, the situation could easily be recreated.
Discussion:	Currently, there is no explicit notation for structured loops. Neither is there direct support through reconfiguration functions. However, through a combination of a Flow Junction, Flow Merge and a variable in the workflow ontology, this pattern can be simulated. The structured loop is therefore a shorthand notation that has no sufficient technical reason for implementation in StPowla.

Recursion:

Description:	An activity is able to call itself.
Support:	No - StPowla allows the design of workflows to include instances of tasks, where two or more instances of the same task are independent from one another. No task is able to call itself.
Discussion:	The implementation of recursion is potentially complex, leading to continuous loops if the design is incorrect. At a business level, recursive functionality can be defined as a single unit of functionality, with the recursive aspect an implementation detail. A task may be implemented using some recursion, but there is little need for it at the workflow level. Thus there is insufficient technical requirement for implementing recursion in StPowla.

7.4.7 Termination Patterns

Implicit Termination:

Description:	A given process terminates if there is no further work item to process.
Support:	No - all StPowla workflows must terminate in a single end point (with the shorthand exception of the Exceptional Delete).
Discussion:	This pattern can lead to sloppy workflow design. It would be clearer and more precise if it was made explicit that, should no more tasks be available for processing, the next workflow item is the end point. Even if there are tasks still to be executed, a deletion reconfiguration can remove them prior to completing the workflow. At this time there is no sufficient reason to support this workflow pattern.

Explicit Termination:

Description:	A process instance terminates when it reaches a nominated state, typically identified by an end node.
Support:	Partial - StPowla provides support through having a single end operator on each workflow and reconfiguration functions to ensure it is the next item reached.
Discussion:	Although StPowla provides end nodes to workflows, it does not permit two or more branches to exist where one branch goes to the end node whilst aborting or failing the remaining tasks in other branches. Instead, a Conditional Merge operator will be present to enforce completion of remaining work before terminating the workflow. The policy designer must be aware of the position in the workflow where the workflow may explicitly terminate in order to know what remaining tasks may exist. This implies that for each point at which an explicit termination is possible, a policy may be required to delete and/or abort other tasks.

7.4.8 Trigger Patterns

Transient Trigger:

Description:	A pre-defined event from the workflow or the external environment triggers an existing activity into action.
Support:	Partial - no direct support is provided, but a policy can insert a copy of an existing task into the current workflow location.
Discussion:	StPowla permits an event to invoke a policy, which in turn invokes a reconfiguration function. If that function were to insert a task into the workflow's current (i.e. next) active position, the effect is similar to this workflow pattern. At a specific event, under defined conditions, a new task can be executed.

Persistent Trigger:

Description:	This is similar to the transient trigger, except that the workflow does not need to execute the handling activity immediately. The signal is retained until the workflow reaches the handling activity.
Support:	Partial - no direct support is provided, but a policy can insert a copy of an existing task into the current workflow location.
Discussion:	As per the Transient Trigger, StPowla permits the insertion of a new task into the workflow. A significant difference is that the policy/workflow combination permits the user to insert a task almost anywhere inside the workflow. Thus it can be argued that StPowla provides support for both these trigger patterns through the same policy usage.

7.5 Policy Conflict

One significant area of research not covered by this thesis is on policy conflict. If policies are defined as “*information which can be used to modify the behaviour of a system*” [59], then policy conflict can be accurately defined as the situation when two or more policies whose information instructs a system to modify its behaviour in incompatible ways.

Policy conflict is an example of the wider problem area known as feature interaction [74]: if a feature f_1 satisfies a property ϕ_1 (written $f_1 \models \phi_1$), and $f_2 \models \phi_2$; a feature interaction is said to occur if, when the features are composed (denoted $f_1 \oplus f_2$) we do not have $f_1 \oplus f_2 \models \phi_1 \wedge \phi_2$ [41].

For example, in the world of telecoms, a user cannot completely enable both features *call waiting* and *call forward when busy* simultaneously without conflict. The first feature holds incoming calls until the current call has finished whereas the second forwards the incoming call to another destination. Clearly, both are incompatible.

Business processes can also be considered in the domain of feature interaction. More specifically, if a business process is subject to policies that can potentially define process variability, the potential for feature interaction problems significantly increases. For example, conflicts can occur between refinement policies (i.e. setting conflicting properties), reconfiguration rules (i.e. setting conflicting actions) and even service selection criteria. For more information, we refer the interested reader to [41].

In StPowla, policy conflict is a significant area of research on its own and is thus not the focus of this thesis. Instead, for more information the reader is encouraged to read [60], which is collaborative work.

7.6 Business Value

The final evaluation perspective to consider is that of business value. All throughout this thesis, we have emphasized StPowla as a solution aimed at a more abstract level of business design, rather than technical implementation. As such, the connection between tasks and services has been hypothetical and all discussion has been kept inside the business domain.

Nevertheless, it is important to understand that there is a very real business need for a solution like StPowla. The integration with Service Oriented Architecture has been widely acknowledged as becoming part of the very fabric of business, rather than just a tool that business uses[2]. We have already identified a simple procurement example in the first chapter and possible variations. One may consider a number of other situations that can frequently occur, such as in Ebay trading where the normal process ends in the buyer providing feedback on the seller to close the process. If the buyer has any reason to object to the business transaction (e.g. the seller did not send the item or the item was not as described), then the buyer has the option to open a *case* with Ebay, who will investigate any claims. This is clearly an extension to the normal process, but could also be seen as a distinct process in itself.

Let us consider another business scenario. ATX Technologies[3] is a company that specialises in providing innovative tools for the automated migration of legacy applications to more modern, agile environments, such as .NET and Java. The aim is always to generate code according to the best practices of the target environment whilst maximising the automatic conversion rate, noting that a number of situations may arise wherein some manual work may be required on a small proportion of the generated code. In particular, ATX's product Forms2Net [5] automates the transformation of Oracle Forms[4] applications to Microsoft .NET, with a conversion rate typically between 80 and 100%. It is on

[2] See article "IBM has high hopes for 'Next Big Thing' in software", Times Online, 3 April 2006

[3] `http://www.atxtechnologies.co.uk`

[4] http://www.oracle.com/technetwork/developer-tools/forms/overview/index.html

this product that we will focus.

During the Forms2Net pre-sales process, an assessment is frequently carried out on a prospect's Oracle Forms application. The process involves using a tool known as the Forms2Net Analyzer to extract statistics from each module of the application. These statistics indicate the size and complexity of each module, for example recording the number of data blocks, items, triggers, windows, canvases, lines of code, database queries, etc.

With the results from the Forms2Net Analyzer, an ATX consultant will upload the statistics to an internal server in order to generate two effort estimation reports. The first report indicates the structural complexity of each module and the second indicates the migration complexity. The structural complexity of each module, combined with the number of lines of embedded PL/SQL code provides a means of estimating the amount of time it could take to rewrite the application by hand to the chosen target environment. The migration complexity provides a means of estimating how much time it could take to perform the stage known as *code completion*. This stage is the manual work required after the automatic conversion to finish the application prior to testing.

The process of generating a project assessment report is relatively straightforward. It involves ascertaining the structural and migration complexities and inserting the resulting data into a report generator that provides a document suitable for customers. However, there are situations in which the process is not straightforward.

The structural complexity of each module is important as it helps understand how long it could take to recreate an equivalent structure in .NET by hand, and also provides initial estimations as to the duration required for preparing new test scripts and the amount of time required for testing. However, Oracle Forms applications can exist in many different sizes, from 1 module to over 10,000 modules and potentially more[5]. The current esti-

[5] From ATX's experience it is difficult to quantify the size of the average Oracle Forms application since it depends on the business problem the application addresses and the design in which it was made. However, a rough estimate suggests that an average size application is between 200 and 500 modules.

mation mechanism is though unable to provide accurate estimations for applications of extremely small size and those of extremely large size. The reason for this is that Oracle Forms provides significant functionality at runtime as part of the environment. For example, connections to the database are provided and managed for free. A set of in-built functions are available to the developer and a number of triggers are predefined to react to specific events. All these aspects contribute towards an application infrastructure that would need to be recreated accordingly to have a perfectly migrated application in .NET.

The issue is that small applications need at least the same framework as the larger applications. Thus, the work to recreate the framework is the same regardless of application size. Therefore, the structural complexity of each module may vary from small projects to large projects as the overhead time required for recreating the infrastructure is shared per module.

Therefore, when situations arise that an extremely small application is analysed, an ATX consultant must examine the results carefully and decide whether they need to be revised. Likewise, for extremely large projects, estimations will also need to be revised, for example to accommodate additional development work to Forms2Net to improve the conversion rate and thus decrease the amount of code completion required.

So even in this real life situation, a core workflow sometimes needs revision based on exceptional circumstances. Despite a core workflow being easy to follow and mostly automated, circumstances define whether or not this workflow can be rigidly adhered to, or modified to suit a specific situation. It is highly likely that numerous other business situations exist in which a core workflow is not enough to model every single situation. Therefore, some means of expressing variability, as we have demonstrated with StPowla, is a valuable addition to normal business processes.

7.7 Summary

In this section, we have taken an objective view of the role that StPowla, and in particular the reconfiguration functions, can play in the business domain. We have compared StPowla notation and the reconfiguration functions to workflow patterns, using them as the benchmark for expressibility.

We have also discussed the business value of StPowla and specifically the need for variability in exceptional circumstances. As we briefly discussed in the opening chapter, workflows are constrained by design whereas human though is not generally limited to the design and thus there always exists scope for variability for even the most standard processes.

Thus we can conclude that StPowla has both a business value and a good technical match to the requirements of workflow modelling. It is not expected for StPowla to be able to model all workflow control flow patterns due to the difference in perspective, remembering that StPowla is targeted towards business users whereas workflow patterns are generally more technical in nature. The most notable constraint is the lack of support for multiple instance patterns, however they are less relevant in the context of this thesis.

More crucially, the reconfiguration functions we have described provide a powerful extension to standard StPowla notation, allowing better support for workflow patterns in general, through the capability of allowing wholesale changes to a workflow from a even a small set of functions. The notation itself provides little more than other workflow languages or notations, other than simplicity and domain suitability. However the addition of the reconfiguration functions allows the dynamic change we have been seeking.

Chapter 8

Conclusion

8.1 Introduction

We now return to the situation presented in Chapter 1. A simple procurement scenario consists of a supplier receiving an order, collecting items, packing items, shipping the package to the customer and invoicing them. Under most circumstances, this process is correct, although there may be some circumstances in which it is desirable to modify the process for that particular executing instance. The example we gave was where an incoming order had a substantially high value and the supplier desired to charge the customer a deposit first.

Our aim was to provide a mechanism whereby business users could define variability over a business-level workflow such that the core process remains the same and is affected only when required. We identified two extremes: Workflow A specifies no tasks at all and the control flow is empty. It is populated with tasks through policy functions at runtime. Workflow B is a significantly complex workflow with every single possible condition identified and embedded into the control flow.

Both situations bring unnecessary complexities. In the first situation, a lot of effort is re-

quired to define a process in abstract terms without seeing the final model, even in a visual model. The potential for conflict between rules which introduce tasks becomes significant due to the possibility of numerous rules being present and active simultaneously. One might also consider concurrency issues when actioning those rules. In the second situation, the workflow remains rigid, even after comprehensive specification. This implies there is no scope for change outside what has already been defined and is thus still representative of the problem we identified.

We wanted a solution that would sit comfortably between these two scenarios, yet at a level abstract from technological implementation that would be suitable for business users. We proposed and presented StPowla as a solution to this problem. A graphical notation provides a visual way of describing workflows. Policies provide a loosely-coupled way of defining variability over workflows. Service Oriented Architecture (SOA) provides the underlying functionality.

We now reflect on the presented work, considering the original research questions to see if they were suitably answered. We proceed to discussing the potential implications of the work at both a technical and business level.

Finally, we consider the limitations of the work presented so far and discuss ideas for future work in order to either address those limitations or extend the existing work. We find that there is rich ground for future research, which does not negate the value of the work done so far, but rather demonstrates the potential impact of StPowla.

8.2 Reflection on Research Questions

In Chapter 1, we highlighted a number of research questions that we were to tackle in this thesis. In this section, we will reflect on the work done to answer those original questions.

Design: We asked first for a suitable method of defining service-targeted workflows. Those are defined at the business level, abstract from implementation details and especially functional implementation. The information included with each workflow activity, known as a task, should extend as far as the requirements for that task. Neither did we look for a means to match services to tasks - this was an assumption made at the beginning.

After a review of business process solutions, we identified three different categories of ways to define workflows: code-based descriptions, notation-based descriptions and calculi-based descriptions. We identified a number of solutions in these categories and elected to provide a basic graphical notation that would be abstract from all current solutions and focus on the core issue of defining the workflow. The StPowla notation includes a relatively small number of operators that permits the organisation of tasks into a flexible and expressive execution schedule, known as the task map (i.e. the workflow). We believe the notation to be highly effective in what it attempts to achieve: the description of a wide range of business processes using a simplified language suitable for the business domain.

We asked second for a means to specifying the variability over those workflows. A review concluded that there were no suitable options available at this level of abstraction, although there were a small number of proposals at the orchestration or logical level. Policies provided a good solution at the business level, with Appel selected as the candidate language due to its makeup of the core language and domain-dependent extension.

We elected to use policies as they fit our purposes well: policies are defined separately to the system which they affect (in keeping with the service oriented nature of this subject) and they are highly expressive.

It was this extension possibility that we used to customize Appel to this domain. Appel also allowed us to define policies as either goals or event-condition-action (ECA) rules, both of which are useful for our purposes. Variability specification was available through the use of functions on the workflow and property modification expressions, both potentially composed through the use of logical operators.

In Chapter 3, we identified both the notation constructs and the policy syntax available. We provided a brief discussion on how workflow tasks map to services (although this was outside the scope of our work) and we provided more information on the Service Level Agreement (SLA) language. We also briefly discussed the pragmatics of the customization, i.e. how one can compose two or more policies on the same workflow task. Thus we have satisfied our research questions on the subject of design.

Scope: We asked three questions on the subject of scope. First, we asked what kind of variability we need and second in what way can the business process be changed. These questions are difficult to answer as the problem resides within human thought. Since humans are able to think "outside the box", i.e. bend the rules, we needed to be capable of defining almost any change on the workflow. Thus, we defined a set of functions that would allow us to make several powerful modifications to the workflow structure. Chapter 4 provides the full description of those functions, which we identified as reconfiguration functions. These descriptions included graph transformation rules to visualize their effects. Those functions enabled us to add any workflow item to an existing workflow, remove any existing item and modify any properties.

Third, we asked what limitations should there be. In the definition of reconfiguration functions, there were restrictions, for example the insertion of a task in parallel with another item could only be done using a predefined split/merge operator. Furthermore, a task could not be inserted in parallel with an operator. The reasons for these constraints though are natural - they maintain the integrity of the workflow.

Our evaluation on the scope of the reconfiguration showed that, compared to workflow patterns which were used as the benchmark, StPowla notation and reconfiguration functions satisfied partially or fully 29 out of a total of 43 patterns. Of the remaining 14 patterns, we determined that the 7 multiple instance patterns were not crucial to our needs and the final 7 each had either a suitable justification for its non-inclusion or a suggestion

for potential inclusion.

Combination: Our final research subject was the combination of the business processes and the variability we wished to apply to them. Firstly we asked how we can achieve the combination in a generalized way. Our solution was to define the reconfiguration functions and visualize them using graph transformation rules in Chapter 4. These rules allowed us to see the reconfiguration function in the context of the workflow that we were considering. Furthermore, the rules allowed us to visualize the result of applying two or more rules to the same workflow in a structured way. Thus the graph transformation rules provided us with a generalization of the reconfiguration approach.

We also asked how we could combine an arbitrary number of variability items on a single business process. Our solution, using policies, was to have each policy explicitly say what target item that policy applied to. The pragmatics of the policy customization (Chapter 3) also permitted the composition of multiple policies in an ordered way.

Finally, we asked what limits the combination. The most obvious limitation is that despite the loosely-coupled nature of policies and workflows, the policy author still needs to know about the workflow before writing a suitable policy. This information is exposed through a workflow ontology and thus the policy author needs access to this ontology.

This combination is analogous to normal service oriented development, in which a developer must already know about the service they wish to consume through a descriptive interface.

We also discussed the idea that business rules can sometimes be overarching concerns, over all other workflows executing within the same environment. Typically, such policies must be goals and not ECA rules. Otherwise, we would expect each workflow to contain the same event for the policy to react to, if indeed the policy was designed to be applicable to each workflow. As it is, goals define constraints such as maximum property values and potentially service selection criteria. Policies can define these using the SLA language

and thus be applicable to multiple workflows.

8.3 Limitations and Further Research

StPowla is still only in an infant stage where the concepts are clearly defined yet no implementation exists to demonstrate clearly the ideas StPowla covers. Furthermore, StPowla is aimed only at the business level, leaving several lower levels of detail to be carefully defined. SRML is the first part of this, but modelling tasks still remain for example between tasks and services, in binding, in configuration and in compensation. In this section, we discuss the limitations of StPowla as a means of providing suggestions of future work.

Event Model: The event model employed by StPowla is relatively simplistic, with events for starting, completing or failing each workflow, task or service. There remains plenty of scope for introducing new events into StPowla, for example:

- When a variable value changes;
- When a workflow is interrupted;
- When a condition test passes or fails;
- When a policy is invoked or completed.

The list of potential new events to embed could be potentially vast and as such we cannot cover all of them here. It is sufficient for us to say that a business user may find more benefit (i.e. the ability to model more business processes or more accurately) if they could apply policies to more events.

Task/Service Transaction Modelling: Any events to do with services are not considered in this thesis as they are out of scope, however it is worthwhile taking these business-level StPowla concepts and establishing deeper technical modelling details, for example using process calculi to model the interactions between tasks and services. The purpose for such modelling is to establish the pattern of transactions between tasks and services, such that modelling at the SRML level and at the StPowla level shall be more precise.

Workflow System Support: We argued previously that StPowla concepts at the workflow level were notation or language agnostic. Any existing workflow system could be adapted to introduce StPowla concepts, namely the event model, policy handler and workflow reconfiguration function (based on graph transformation). As such, a review of existing workflow systems, similar to what is found in [78] for workflow patterns, would provide a suitable picture of the applicability of StPowla to real workflow management systems.

Compensation: Currently, StPowla does not provide support for rollback and compensation. At a stretch, StPowla could be said to support compensation tasks through the use of Abort and Fail output paths from tasks, although there is still no support for rollback. Thus, a workflow cannot undo any task it has completed thus far. The use of rollback has significant value due to the number of situations it can be used in, for example:

- Cancelling an order if the delivery date is too late;
- Withdrawing a service request if the cost is too high;
- Cancelling activities due to extraordinary circumstances.

The issue of compensation and rollback is not trivial. Consider that two parallel branches are prematurely cancelled and their compensation actions are invoked. After they have completed, any previous task in the control flow may also need to have its compensation

action invoked. This requires tracking of the exact control flow path taken (including chosen output paths from Flow Junctions, Strict Preferences and Random Choices), and potentially also requires partial compensation for tasks that were started but not completed and for tasks that were completed but whose results were not required (in the case of Random Choice).

Similarly, suppose a task fails in one of two or more parallel branches and its compensation action is invoked. The remaining question is what happens to the other branches that are executing successfully? In this vein, a new strand of research for StPowla is proposed, using the work of [18] as a suitable starting point.

8.4 Beyond SOA

One significant aspect of this work that came apparent during its conception is that it is not limited to SOA. Although SOA provided the backdrop and initial motivation for the work, we could easily remove SOA and still have a valid means to separating core functionality from variability. This is an important aspect to identify with, because it widens the applicability of our work to further scenarios.

For example, a Workflow Management System may make use of local services (not necessarily Web Services) or user-driven tasks. Neither require SOA. Runtime reconfiguration of systems is already available for autonomic ("self-healing") systems and the potential is also there for product line technology. This thesis though applies the concept of runtime modification to processes in a mostly-generic way. The specific implementation details we used were those of policies and SOA.

In fact, one could easily consider that it is rare for processes not to require some form of human involvement. Suppose a form has been received in an administration office (e.g. insurance office), which needs to be processed. The logical first step would be to have

someone enter details from the form into a database, then perform whatever actions are necessary. However, one might want to consider that the real first step is not the processing of the form, but the receipt of it. No system needs to be invoked when receiving mail, nor is any service a requirement when entering data into a system (assuming that the data entry system is not Web Service-based[1].

The point being made here is that several actions can be classed as tasks in our workflow, yet not all of them require a service to implement the required functionality. At the extreme, no services are used to implement the workflow. All tasks could be user-oriented and potentially no computation is required anywhere. But the principle of separation of core functionality and process variability still stands.

8.5 Summary

As a final conclusion, we return to the issue of inevitable choice. The problem of choice is that we might not know all the potential options available to us. In the context of business processes, this implies that we may never be able to appropriately define a workflow completely that describes the business process for the simple reason that many possible options of variability may be required. Thus there is a definite need for being able to naively define a core process with the option of defining variability later on and distinct from that process.

StPowla offers a solution in a service oriented context such that the promise of truly dynamic applications can be realised through having dynamic workflows. Although still in its early stages, StPowla notation and reconfiguration functions already provide a solution to this problem. In conjunction with the evaluation, there are areas which can be improved upon, namely support for new workflow patterns, including those relating to

[1]Even still, this would not change our argument that a user manually entering data is an interactive task, not fully automated. Even using Optical Character Recognition requires a user to set the document in the correct feeder tray and then initiate the scan.

data and exception handling, plus improved scope for workflow intervention and compensation actions.

In the future, as business seeks further automation in more areas to improve operational efficiency and maintain competitiveness, we strongly believe that StPowla and the ideas it encapsulates can have a significant impact.

Bibliography

[1] João Abreu, Laura Bocchi, José Luiz Fiadeiro, and Antónia Lopes. Specifying and Composing Interaction Protocols for Service-Oriented System Modelling. In John Derrick and Jüri Vain, editors, *FORTE*, volume 4574 of *Lecture Notes in Computer Science*, pages 358–373. Springer, 2007.

[2] João Abreu and José Luiz Fiadeiro. A Coordination Model for Service-Oriented Interactions. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

[3] Michael Adams, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4275 of *Lecture Notes in Computer Science*, pages 291–308. Springer, 2006.

[4] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraiu. *Web Services: Concepts, Architectures and Applications*. Springer, 2004.

[5] Luis Filipe Andrade, João Gouveia, Miguel Antunes, Mohammad El-Ramly, and Georgios Koutsoukos. Forms2Net - Migrating Oracle Forms to Microsoft .NET. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *GTTSE*, volume 4143 of *Lecture Notes in Computer Science*, pages 261–277. Springer, 2006.

[6] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, Ivana Trickovic, and Sinisa Zimek. Web Services Choreography Interface (WSCI). W3C, Aug 2002. http://www.w3.org/TR/wsci/.

[7] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, Ventakash Chopella, Kannan Govindarajan, Alan Karp, Harumi Kuno, Mike Lemon, Gregory Pogossiants, Shamik Sharma, and Scott Williams. Web Services Conversation Language (WSCL). W3C, Mar 2002. http://www.w3.org/TR/wscl10/.

[8] Tom Bellwood, Steve Capell, Luc Clement, John Colgraveand, Matthew J. Dovey, Daniel Feygin, Andrew Hately, Rob Kochman, Paul Macias, Mirek Novotny, Massimo Paolucci, Claus von Riegen, Tony Rogers, Katia Sycara, Pete Wenzel, and Zhe Wu. Universal Description Discovery & Integration Technical Specification Version 3.0.2. http://uddi.org/pubs/uddi_v3.htm, 10 2004. Accessed on 30 April 2011.

[9] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. A Foundational Vision of e-Services. In Christoph Bussler, Dieter Fensel, Maria E. Orlowska, and Jian Yang, editors, *WES*, volume 3095 of *Lecture Notes in Computer Science*, pages 28–40. Springer, 2003.

[10] Laura Bocchi, José Luiz Fiadeiro, and Antónia Lopes. A Use-Case Driven Approach to Formal Service-Oriented Modelling. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 155–169. Springer, 2008.

[11] Laura Bocchi, José Luiz Fiadeiro, and Antónia Lopes. Service-Oriented Modelling of Automotive Systems. In *COMPSAC*, pages 1059–1064. IEEE Computer Society, 2008.

[12] Laura Bocchi, Stephen Gorton, and Stephan Reiff-Marganiec. Engineering Service Oriented Applications: From StPowla Processes to SRML Models. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2008.

[13] Laura Bocchi, Stephen Gorton, and Stephan Reiff-Marganiec. From StPowla processes to SRML models. *Formal Asp. Comput.*, 22(3-4):243–268, 2010.

[14] Laura Bocchi, Yi Hong, Antónia Lopes, and José Luiz Fiadeiro. From BPEL to SRML: A Formal Transformational Approach. In Dumas and Heckel [24], pages 92–107.

[15] Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro. A Calculus for Long-Running Transactions. In Elie Najm, Uwe Nestmann, and Perdita Stevens, editors, *FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2003.

[16] Mario Bravetti and Gianluigi Zavattaro. Service Oriented Computing: A New Challenge for Process Algebras. *Electr. Notes Theor. Comput. Sci.*, 162:121–125, 2006.

[17] Christoph Bussler, Richard Hull, Sheila A. McIlraith, Maria E. Orlowska, Barbara Pernici, and Jian Yang, editors. *Web Services, E-Business, and the Semantic Web, CAiSE 2002 International Workshop, WES 2002, Toronto, Canada, May 27-28, 2002, Revised Papers*, volume 2512 of *Lecture Notes in Computer Science*. Springer, 2002.

[18] Luís Caires, Carla Ferreira, and Hugo Torres Vieira. A Process Calculus Analysis of Compensations. In Christos Kaklamanis and Flemming Nielson, editors, *TGC*, volume 5474 of *Lecture Notes in Computer Science*, pages 87–103. Springer, 2008.

[19] Fabio Casati, Eric Shan, Umeshwar Dayal, and Ming-Chien Shan. Business-oriented management of Web services. *Commun. ACM*, 46(10):55–60, 2003.

[20] Anis Charfi and Mira Mezini. AO4BPEL: An Aspect-oriented Extension to BPEL. In *World Wide Web*, pages 309–344, 2007.

[21] Jean-Philip Pritchard Colin Armistead and Simon Machin. Strategic Business Process Management for Organisational Effectiveness. *Long Range Planning*, 32(1):96–106, March 1999.

[22] Andrea Corradini and Reiko Heckel. Graph Transformation and Visual Modeling Techniques: Workshop Summary and HowTo. *Bulletin of the EATCS*, 72:69–76, 2000.

[23] Carine Courbis and Anthony Finkelstein. Towards an Aspect Weaving BPEL engine. In Y. Coady and D. H. Lorenz, editors, *the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Lancaster, United Kingdom, March 2004.

[24] Marlon Dumas and Reiko Heckel, editors. *Web Services and Formal Methods, 4th International Workshop, WS-FM 2007, Brisbane, Australia, September 28-29, 2007. Proceedings*, volume 4937 of *Lecture Notes in Computer Science*. Springer, 2008.

[25] Marlon Dumas and Arthur H. M. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In Martin Gogolla and Cris Kobryn, editors, *UML*, volume 2185 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2001.

[26] Rik Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, University of Twente, 2002.

[27] Rik Eshuis and Roel Wieringa. A Formal Semantics for UML Activity Diagrams. Technical Report TR-CTIT-01-04, Centre for Telematics and Information Technology, University of Twente, 2001.

[28] Rik Eshuis and Roel Wieringa. A Real-Time Execution Semantics for UML Activity Diagrams. In Heinrich Hußmann, editor, *FASE*, volume 2029 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2001.

[29] José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. A Formal Approach to Service Component Architecture. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 193–213. Springer, 2006.

[30] José Luiz Fiadeiro, Antónia Lopes, Laura Bocchi, and João Abreu. A Formal Approach to Service-Oriented Modelling. University of Leicester, 2009. A comprehensive account of SRML.

[31] National Institute for Standards and Technology. `http://csrc.nist.gov/groups/SNS/cloud-computing/index.html`. accessed 15 April 2011.

[32] Organisation for the Advancement of Structured Information Standards. Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`, April 2007. Accessed on 16 April 2011.

[33] Xiang Fu, Tevfik Bultan, and Jianwen Su. Formal Verification of e-Services and Workflows. In Bussler et al. [17], pages 188–202.

[34] Stefania Gnesi and Franco Mazzanti. A model checking verification environment for UML statecharts. In *AICA, Udine 2005*, 5-7 October 2005.

[35] Nicolas Gold, Andrew Mohan, Claire Knight, and Malcolm Munro. Understanding Service-Oriented Software. *IEEE Software*, 0740-7459(04):71–77, March/April 2004.

[36] Stephen Gorton, Carlo Montangero, Stephan Reiff-Marganiec, and Laura Semini. StPowla: SOA, Policies and Workflows. In Nitto and Ripeanu [63], pages 351–362.

[37] Stephen Gorton and Stephan Reiff-Marganiec. Policy Support for Business-oriented Web Service Management. In J. Alfredo Sánchez, editor, *LA-WEB*, pages 199–202. IEEE Computer Society, 2006.

[38] Stephen Gorton and Stephan Reiff-Marganiec. Towards a Task-Oriented, Policy-Driven Business Requirements Specification for Web Services. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 465–470. Springer, 2006.

[39] Stephen Gorton and Stephan Reiff-Marganiec. Towards a Task-Oriented, Policy-Driven Business Requirements Specification for Web Services. Research Reports in Computer Science 009, University of Leicester, 2006.

[40] Stephen Gorton and Stephan Reiff-Marganiec. Policy-driven Business Management over Web Services. In *Integrated Network Management*, pages 721–724. IEEE, 2007.

[41] Stephen Gorton and Stephan Reiff-Marganiec. Towards Feature Interactions in Business Processes. In Lydie du Bousquet and Jean-Luc Richier, editors, *ICFI*, pages 99–113. IOS Press, 2007.

[42] Business Rules Group. Defining Business Rules - What Are They Really? `http://www.businessrulesgroup.org/first_paper/br01c0.htm`, July 2000. Accessed on 16 April 2011.

[43] Claus Hagen and Gustavo Alonso. Exception Handling in Workflow Management Systems. *IEEE Trans. Software Eng.*, 26(10):943–958, 2000.

[44] Rachid Hamadi and Boualem Benatallah. A Petri Net-based Model for Web Service Composition. In Klaus-Dieter Schewe and Xiaofang Zhou, editors, *ADC*, volume 17 of *CRPIT*, pages 191–200. Australian Computer Society, 2003.

[45] Michael Hammer. Reengineering Work: Don't Automate, Obliterate. *Harvard Business Review*, pages 104–112, July/August 1990.

[46] Reiko Heckel. A Formal Approach to Service Specification and Matching based on Graph Transformation. 4th International Workshop on Web Services and Formal Methods WSFM-2004, 2004. Included in slides based on a presented paper.

[47] Reiko Heckel. Stochastic Analysis of Graph Transformation Systems: A Case Study in P2P Networks. In Dan Van Hung and Martin Wirsing, editors, *Proc. Intl. Colloquium on Theoretical Aspects of Computing (ICTAC'05), Hanoi, Vietnam*, volume 3722. Springer-Verlag, October 2005. Invited paper.

[48] Sebastian Hinz, Karsten Schmidt 0004, and Christian Stahl. Transforming BPEL to Petri Nets. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management*, volume 3649, pages 220–235, 2005.

[49] Hugo Haas and Allen Brown. Web Services Glossary. W3C Working Group Note, World Wide Web Consortium (W3C), 2004. `http://www.w3.org/TR/ws-gloss/`.

[50] IBM. IBM Solutions Grid for Business Partners. `http://joung.im.ntu.edu.tw/teaching/distributed_systems/documents/IBM_grid_wp.pdf`, March 2002. accessed 15 April 2011.

[51] Ivar Jacobson, Grady Booch, and James Rambaugh. *The Unified Software Development Process*. Addison-Wesley Longman, 1998.

[52] Faouzi Kamoun. A Roadmap towards the Convergence of Business Process Management and Service Oriented Architecture. *Ubiquity*, 2007(April):1–1, 2007.

[53] Dimka Karastoyanova and Frank Leymann. BPEL'n'Aspects: Adapting Service Orchestration Logic. In *ICWS*, pages 222–229. IEEE, 2009.

[54] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[55] Vadim Kotov. Towards Service-Centric System Organization. Technical Report HPL-2001-54, Hewlett Packard, 2001. Accessed on 15 April 2011.

[56] Akhil Kumar and J. Leon Zhao. Workflow support for electronic commerce applications. *Decision Support Systems*, 32(3):265–278, 2002.

[57] Frank Leymann. Web Service Flow Language. Technical report, IBM Software Group, 2001. `http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.

[58] Antónia Lopes, José Luiz Fiadeiro, and Laura Bocchi. Algebraic Semantics of Service Component Modules. In *Algebraic Development Techniques*, volume 4409 of *Lecture Notes in Computer Science*, pages 37–55. Springer, 2007.

[59] Emil Lupu and Morris Sloman. Conflicts in Policy-Based Distributed Systems Management. *IEEE Trans. Software Eng.*, 25(6):852–869, 1999.

[60] Carlo Montangero, Stephan Reiff-Marganiec, and Laura Semini. Logic-based Conflict Detection for Distributed Policies. *Fundam. Inform.*, 89(4):511–538, 2008.

[61] Robert Müller, Ulrike Greiner, and Erhard Rahm. $A_{gent}w_{ork}$: a workflow system supporting rule-based workflow adaptation. *Data Knowl. Eng.*, 51(2):223–256, 2004.

[62] Surya Nepal, Alan Fekete, Paul Greenfield, Julian Jang, Dean Kuo, and Tony Shi. A service-oriented workflow language for robust interacting applications. In Robert Meersman, Zahir Tari, Mohand-Said Hacid, John Mylopoulos, Barbara Pernici, Özalp Babaoglu, Hans-Arno Jacobsen, Joseph P. Loyall, Michael Kifer, and Stefano Spaccapietra, editors, *OTM Conferences (1)*, volume 3760 of *Lecture Notes in Computer Science*, pages 40–58. Springer, 2005.

[63] Elisabetta Di Nitto and Matei Ripeanu, editors. *Service-Oriented Computing - ICSOC 2007 Workshops, International Workshops, Vienna, Austria, September 17, 2007, Revised Selected Papers*, volume 4907 of *Lecture Notes in Computer Science*. Springer, 2009.

[64] Jasmine Noel. BPM and SOA: Better Together. IBM White Paper, IBM Corporation, 2005. `http://www.findwhitepapers.com/docs/vendors/IBM/BPM\%20SOA\%20BetterTogether\%20-\%20Offer.pdf`.

[65] Object Management Group (OMG). *Business Process Model and Notation (BPMN) Specification*, August 2009. http://www.omg.org/spec/BPMN/Current.

[66] David O'Riordan. Business Process Standards for Web Services. `http://www.webservicesarchitect.com/content/articles/oriordan01.asp`. Accessed on 16 April 2011.

[67] Justin O'Sullivan, David Edmond, and Arthur H. M. ter Hofstede. What's in a Service? *Distributed and Parallel Databases*, 12(2/3):117–133, 2002.

[68] Chun Ouyang, Marlon Dumas, Arthur H. M. ter Hofstede, and WilM. P. van der Aalst. From BPMN Process Models to BPEL Web Services. In *ICWS*, pages 285–292. IEEE Computer Society, 2006.

[69] Flavio De Paoli, Andrea Maurino, Ioan Toma, Justin O'Sullivan, Marcel Tilly, and Glen Dobson. Non-Functional Properties and Service Level Agreements in Service-Oriented Computing (NFPSLA-SOC '07) - Organizers' Workshop Summary. In Nitto and Ripeanu [63], pages 43–44.

[70] Maja Pesic and Wil M. P. van der Aalst. A Declarative Approach for Flexible Business Processes Management. In Johann Eder and Schahram Dustdar, editors, *Business Process Management Workshops*, volume 4103 of *Lecture Notes in Computer Science*, pages 169–180. Springer, 2006.

[71] Jan Recker and Jan Mendling. On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In Thiband Latour and Michaël Petit, editors, *EMMSAD*, pages 521–532. IEEE Computer Society, 2006.

[72] Manfred Reichert and Peter Dadam. ADEPT$_{\text{flex}}$-Supporting Dynamic Changes of Workflows Without Losing Control. *J. Intell. Inf. Syst.*, 10(2):93–129, 1998.

[73] Stephan Reiff-Marganiec and Kenneth J. Turner. Use of Logic to Describe Enhanced Communications Services. In Doron Peled and Moshe Y. Vardi, editors, *FORTE*, volume 2529 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2002.

[74] Stephan Reiff-Marganiec and Kenneth J. Turner. Feature Interaction in Policies. *Computer Networks*, 45(5):569–584, 2004.

[75] Stephan Reiff-Marganiec, Kenneth J. Turner, and Lynne Blair. APPEL: The ACCENT Policy Environment/Language. Technical Report CSM-164, University of Stirling, Jun 2005.

[76] Dirk Riehle and Heinz Züllighoven. Understanding and Using Patterns in Software Development. *TAPOS*, 2(1):3–13, 1996.

[77] Florian Rosenberg and Schahram Dustdar. Business Rules Integration in BPEL - A Service-Oriented Approach. In *CEC*, pages 476–479. IEEE Computer Society, 2005.

[78] Nick Russell, Arthur H.M. ter Hofstede, Wil M.P. van der Aalst, and Nataliya Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical Report BPM-06-22, BPM Center, 2006.

[79] Wasim Sadiq, Olivera Marjanovic, and Maria E. Orlowska. Managing Change and Time in Dynamic Workflow Processes. *Int. J. Cooperative Inf. Syst.*, 9(1-2):93–116, 2000.

[80] Christian Stahl. A Petri Net Semantics for BPEL. Technical report, University, 2004.

[81] Stefano Bistarelli and Ugo Montanari and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.

[82] Christian Stefansen. SMAWL: A SMAll Workflow Language Based on CCS. In Oscar Pastor and João Falcão e Cunha, editors, *CAiSE Short Paper Proceedings*, volume 3520 of *Lecture Notes in Computer Science*. Springer, 2005.

[83] Harald Störrle. Semantics of Control-Flow in UML 2.0 Activities. In *VL/HCC*, pages 235–242. IEEE Computer Society, 2004.

[84] Harald Störrle and Jan Hendrik Hausmann. Towards a Formal Semantics of UML 2.0 Activities. In Peter Liggesmeyer, Klaus Pohl, and Michael Goedicke, editors, *Software Engineering*, volume 64 of *LNI*, pages 117–128. GI, 2005.

[85] Stefan Tai, Rania Khalaf, and Thomas A. Mikalsen. Composition of Coordinated Web Services. In Hans-Arno Jacobsen, editor, *Middleware*, volume 3231 of *Lecture Notes in Computer Science*, pages 294–310. Springer, 2004.

[86] Satish Thatte. XLANG. Microsoft Corporation. http://www.gotdotnet.com/team/xml$_$wsspecs/xlang-c/default.htm.

[87] Kenneth J. Turner. The Accent Policy Wizard. Technical Report CSM-166, University of Stirling, December 2005.

[88] Moe Thandar Tut and David Edmond. The Use of Patterns in Service Composition. In Bussler et al. [17], pages 28–40.

[89] Wil M. P. van der Aalst, Alistair P. Barros, Arthur H. M. ter Hofstede, and Bartek Kiepuszewski. Advanced Workflow Patterns. In Opher Etzion and Peter Scheuermann, editors, *CoopIS*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer, 2000.

[90] Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Web Service Composition Languages: Old Wine in New Bottles? In *EUROMICRO*, pages 298–307. IEEE Computer Society, 2003.

[91] Wil M P van der Aalst and Stefan Jablonski. Dealing with Workflow Change: Identification of Issues and Solutions. *International Journal of Computer Systems Science and Engineering*, 15(5):267–276, September 2000.

[92] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Inf. Syst.*, 30(4):245–275, 2005.

[93] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[94] Aad P. A. van Moorsel. Ten-Step Survival Guide for the Emerging Business Web. In Bussler et al. [17], pages 1–11.

[95] Dániel Varró. Towards Symbolic Analysis of Visual Modeling Languages. *Electr. Notes Theor. Comput. Sci.*, 72(3), 2003.

[96] Asir S Vedamuthu, David Orchard, Frederick Hirsch, Maryann Hondo, Prasad Yendluri, Toufic Boubez, and Umit Yalconalp. Web Services Policy 1.5 - Framework. http://www.w3.org/TR/ws-policy/, September 2007. Accessed on 16 April 2011.

[97] Harry Jiannan Wang. *A Logic-based Methodology for Business Process Analysis and Design: Linking Business Policies to Workflow Models*. PhD thesis, University of Arizona, 2006.

[98] Stephen A. White. Using BPMN to model a BPEL Process. *BPTrends*, 2005. http://www.bptrends.com, accessed on 15/03/06.

[99] Arthur H.M. ter Hofstede Wil M.P. van der Aalst, Marlon Dumas and Petia Wohed. Pattern-Based Analysis of BPML (and WSCI). Technical Report FIT-TR-2002-05, Queensland University of Technology, 2002.

[100] Martin Wirsing, Laura Bocchi, Allan Clark, José Luiz Fiadeiro, Stephen Gilmore, Matthias Hölzl, Nora Koch, and Rosario Pugliese. *SENSORIA: Engineering for Service-Oriented Overlay Computers*. MIT, June 2007.

[101] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In Il-Yeol Song, Stephen W. Liddle, Tok Wang Ling, and Peter Scheuermann, editors, *ER*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2003.

[102] Dong Yang and Shen sheng Zhang. Using π-calculus to Formalize UML Activity Diagrams. In *ECBS*, pages 47–54. IEEE Computer Society, 2003.

[103] Hong Qing Yu and Stephan Reiff-Marganiec. A Method for Automated Web Service Selection. In *SERVICES I*, pages 513–520. IEEE Computer Society, 2008.

[104] Hong Qing Yu and Stephan Reiff-Marganiec. A Backwards Composition Context Based Service Selection Approach for Service Composition. In *IEEE SCC*, pages 419–426. IEEE Computer Society, 2009.

[105] Hong Qing Yu and Stephan Reiff-Marganiec. Automated Context-Aware Service Selection for Collaborative Systems. In Pascal van Eck, Jaap Gordijn, and Roel Wieringa, editors, *CAiSE*, volume 5565 of *Lecture Notes in Computer Science*, pages 261–274. Springer, 2009.