# Object-centric Process Models and the Design of Flexible Processes

by

Guy Matthew Redding

A dissertation submitted for the degree of
IF49 Doctor of Philosophy

Faculty of Science and Technology
Queensland University of Technology
Brisbane, Australia

Principal Supervisor: Prof. Marlon Dumas

Associate Supervisor: Prof. Arthur H.M. ter Hofstede

September 2009

# Certificate of Acceptance

# Statement of Original Authorship

*The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person, except where due reference is made.*

Signed: ________________________________________

Date: ________________________________________

# Acknowledgements

In 2003 I was first exposed to the possibility of a research career while on internship at a large software company. My supervisor at that time once remarked that *"Completing a PhD is the most difficult thing I've ever done"*. Such a thing sounded too easy to say and it stuck with me, since programming seemed to require a greater creative effort than writing. So I was initially fairly sceptical about that. However, if we fast-forward 6 years...

I now fully agree.

My first round of thanks are due to my supervisory team of Marlon Dumas and Arthur ter Hofstede – otherwise known as *Team Martha*. It's been an honour and a privilege to work in a team with such vast knowledge and experience. Your ways of working are amazing to observe and have been simply inspiring for me. My first realisation after our initial meetings was of my tendency towards *mañana* theory and practice, but three years later I hope that the *whole enchilada* is a satisfactory reward for your efforts.

Thanks are also due to the industry partner company for this project, FlowConnect Pty Ltd from Sydney, for providing me with both the opportunity to undertake this study and for access to valuable data and models from the field. I enjoyed both visits to the premises of FlowConnect during this project and I send my best wishes for the future to Adrian Iordachescu, Jarka Sipka, plus the rest of the folks at FlowConnect.

The BPM Group and School of IT at QUT is a vibrant, stimulating place to work due to many smart and fun people there who hail from all over the globe. Thus, from QUT I have made many fulfilling friendships, some of whom are already known as *old mates*. I sincerely hope these friendships will last long into the future. Through a healthy mix of in-house and out-of-office activities (specifically, the coffee group, squash group, Friday evenings group and Mittagessen Gruppe) you guys have helped me through this period of my life. I am extremely grateful to know such people from all over the world as my friends.

I must also thank my immediate family (Phil, Gayle, Gina, Bianca & Carmel) for sticking with me through all these years of study while we've been so many kilometres apart. Guess it's gonna seem mighty strange not having to try to understand any further explanations of how the PhD is going! I love you guys, cheers for every little piece of your support. A special mention also to Bern Healey for his interest in each step of this journey.

I've saved the best until last. Kat, I wouldn't want to have shared the last several years with anybody else – your belief in me has been amazing. Without you in my life it's highly likely that none of this would have been possible. For always being there without question throughout the dizzying highs and terrifying lows of the most difficult thing I've ever done, you thoroughly deserve the ultimate acknowledgement. Although you really are deserving of much, much more!

$\ddot{\smile}$

Guy Redding

15th September, 2009

# Abstract

Mainstream business process modelling techniques promote a design paradigm wherein the activities to be performed within a case, together with their usual execution order, form the backbone of a process model, on top of which other aspects are anchored. This paradigm, while effective in standardised and production-oriented domains, shows some limitations when confronted with processes where case-by-case variations and exceptions are the norm. In this thesis we develop the idea that the effective design of flexible process models calls for an alternative modelling paradigm, one in which process models are modularised along key business objects, rather than along activity decompositions.

The research follows a design science method, starting from the formulation of a research problem expressed in terms of requirements, and culminating in a set of artifacts that have been devised to satisfy these requirements. The main contributions of the thesis are: (i) a meta-model for object-centric process modelling incorporating constructs for capturing flexible processes; (ii) a transformation from this meta-model to an existing activity-centric process modelling language, namely YAWL, showing the relation between object-centric and activity-centric process modelling approaches; and (iii) a Coloured Petri Net that captures the semantics of the proposed meta-model. The meta-model has been evaluated using a framework consisting of a set of workflow patterns. Moreover, the meta-model has been embodied in a modelling tool that has been used to capture two industrial scenarios.

## Keywords

object-oriented, object-centric, process-oriented, activity-centric, workflow systems, process aware information systems, model transformation, workflow patterns, flexibility, flexibility patterns.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Research Area

The flow of work in an organisation is supported by processes that provide a guideline for conducting day-to-day operations, and information systems that serve to automate day-to-day operations. However, the processes that guide an organisation and the information systems (specifically IT systems and applications) that support those processes are commonly mismatched – a situation that has been described as the *Business-IT divide* [95]. These mismatches are often blamed on the lack of understanding or misunderstanding that an organisation has of its IT systems and vice-versa. Closing the divide between business and IT has been an aspiration for some time due to the many purported benefits, which include improved requirement analysis and enactment, more efficient business automation, greater business collaboration (both internally and externally), enhanced organisational competitive advantages, better accountability and higher ongoing IT project success rates.

The discipline of Business Process Management (BPM) has emerged as a paradigm that includes methods, techniques and tools to support the design, documentation, enactment, management and analysis of operational business processes [7]. BPM is both a way of thinking about the workings of a business and a discipline for continuous improvement and management of business process models. BPM is a *holistic* management approach that has been positioned as an *enabler* to progress towards closing the Business-IT divide. However, employing BPM in principle does not completely bridge the Business-IT divide since BPM does not mandate "process-awareness" in IT systems. Recognition of this technological gap has led to the emergence of software suites known as Process-Aware Information Systems (PAIS).

According to the definition given in [27], PAIS are software systems that manage and execute operational processes involving people, applications, and/or information sources on the basis of process models. PAIS are based on technology frameworks that extend BPM for the purpose of enabling business process automation. As shown in Figure 1.1, business process automation is achieved by following the PAIS development lifecycle. The PAIS development lifecycle is a procedure that consists of several phases identified as the design, implementation, enactment and diagnosis phases. Each phase in the lifecycle is related to a number of different tools and systems to support PAIS development and business process automation.

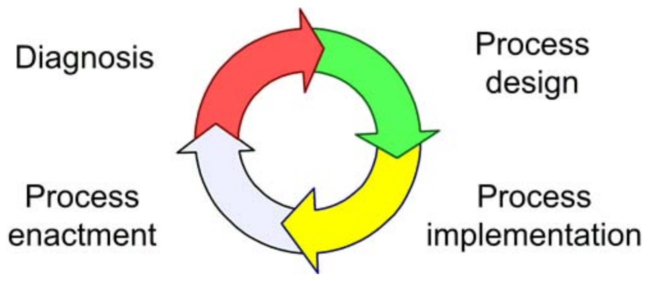


Figure 1.1: The PAIS Development Lifecycle

At the heart of the PAIS development lifecycle are process models. A process model provides a level of abstraction over an information system that makes it possible for a business analyst to specify the operations of a business in terms of what needs to be done (e.g. the tasks in a process). Business operations are captured by process modelling tools, avoiding the need to capture a process with low-level software code. Abstraction away from software-related details of a system into the realm of process models is a highly desirable aspect of process modelling. Some benefits offered by PAIS include an improved ability to react to process change, business performance is easier to measure and the current and/or historical status of work within an organisation can be queried. The trend towards process automation using PAIS technology is exemplified by the developments in process-aware systems and tools by commercial practice. Some examples of such systems and tools include: IBM Websphere MQ Workflow, FLOWer by Pallas Athena, JBoss jBPM and TIBCO iProcess Suite.

To automate a process by means of PAIS, the process must be encoded using a process modelling language. The disadvantage of this explicit encoding is that assumptions are often made concerning how the tasks in a process should be related. These assumptions may turn out to be incorrect and/or not applicable to all cases. It can be argued that this is due to mainstream process modelling techniques. Mainstream process modelling techniques (e.g. BPMN) often promote a *tightly-framed* design paradigm wherein the activities that may be performed within a case, together with their usual execution order, form the backbone of a process on top of which other aspects are anchored. This Fordist paradigm [98], while effective in standardised and production-oriented domains, breaks when confronted with *loosely-framed* or *ad-hoc framed* types of processes in which case-by-case variations, exceptions and user-initiated overrides are the norm. As one might imagine, attempting to automate loosely-framed or ad-hoc processes is a more challenging task than attempting to automate tightly-framed processes, due to the additional concern of balancing process automation with process flexibility.

In Figure 1.2, several categories of PAIS have been arranged according to the degree with which each category of PAIS defines or *frames* a process (e.g. tightly-framed, loosely-framed, ad-hoc framed or unframed) and the nature of process participants of the PAIS (e.g. Process-to-Process, Process-to-Application or Application-to-Application). According to the PAIS classification given in Figure 1.2, workflow technologies can be seen to mostly focus on tightly-framed processes, while less attention is devoted to capturing more loosely-framed types of processes. However there is a perpetual

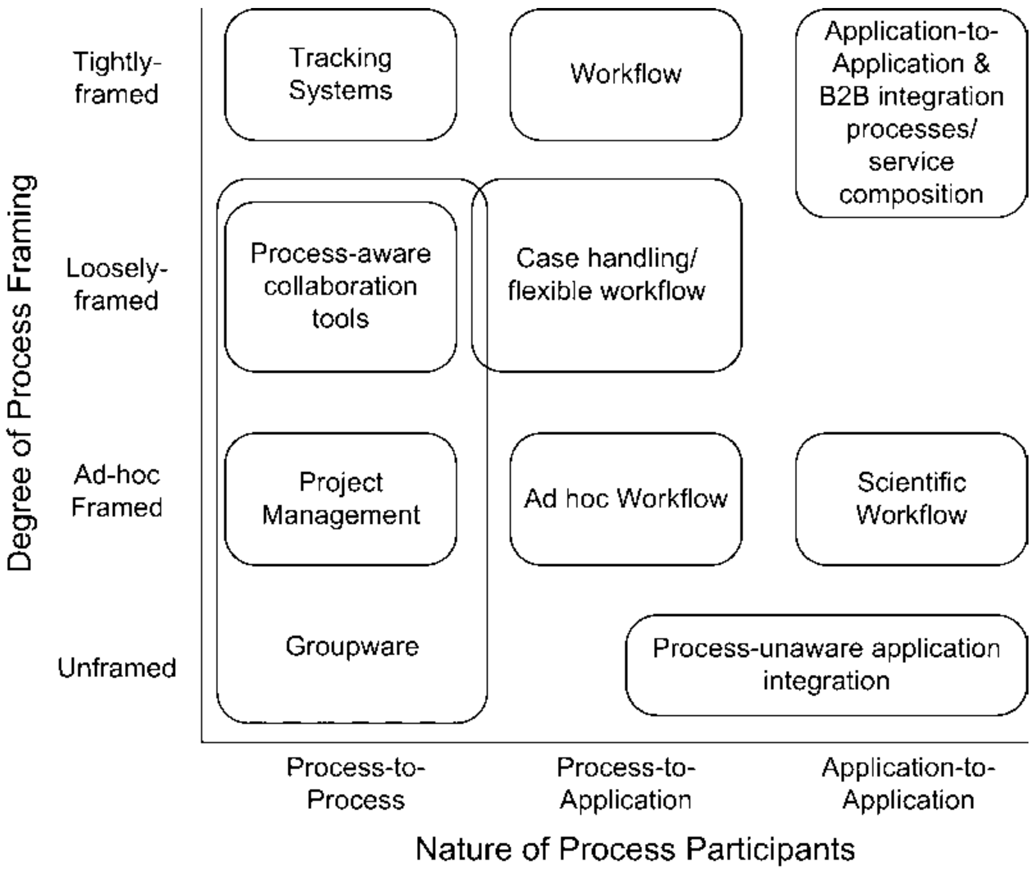

Degree of Process Framing
Tightly-framed
Loosely-framed
Ad-hoc Framed
Unframed
Tracking Systems
Workflow
Application-to-Application & B2B integration processes/ service composition
Process-aware collaboration tools
Case handling/ flexible workflow
Project Management
Ad hoc Workflow
Scientific Workflow
Groupware
Process-unaware application integration
Process-to-Process
Process-to-Application
Application-to-Application
Nature of Process Participants

Figure 1.2: Classification of PAIS Framing (from [27])

need for workflow technologies to cope with processes that are not always tightly-framed. Within the PAIS development lifecycle, a key question is how loosely-framed business process models may be designed to achieve a satisfactory level of both automation and flexibility. Given this question, it can be observed that the effective design and enactment of flexible processes calls for a different approach to encode process models.

Object modelling and process modelling are two examples of established approaches to represent information systems [48] in general and PAIS in particular. Each approach adopts a different perspective and has its own way of thinking. Process models are usually structured in terms of an *activity-centric* approach, which provides a holistic view on the functions (that are executed by activities) in a process and their temporal order of execution to achieve a goal. Meanwhile, modelling an information system in terms of an *object-centric* approach leads to the definition of object types, associations, intra-object behaviour and inter-object interactions. The key difference in an *object-centric* process modelling approach is that the behaviour relevant to an object is captured within a process model, instead of being separated from the process. In this thesis we investigate the possibility of marrying OO modelling techniques with process modelling to define a generic *object-centric* process modelling language.

## 1.2 Problem Statement

It can be observed that in most popular process modelling languages and notations, objects are positioned as second-class citizens in process models. This is evidenced by the situation where objects are placed around the edge of an activity-centric process model. This positioning limits the role that objects play in activity-centric process models, and leads to undesirable outcomes such as forcing a process modeller to maintain two different types of models (activity-based process models in addition to state-based object models). In this thesis, we raise the positioning of objects to first-class citizens by proposing a language that focuses on capturing object lifecycles in process models.

Firstly, we must establish the differences between an activity-centric and an object-centric representation of a process. The most notable difference between these approaches to process modelling lies in identifying the elements that "come first". Using the example given in Figure 1.3, an activity-centric approach primarily consists of a sequence of tasks that belong to a case, e.g. the "Prepare and Send Invitations" case consists of three tasks; a "Prepare Template" task, a "Fill, Print and Pack" multiple instance task and a "Post Invitations" task and state-based object lifecycles that provide input to and accept output from tasks. An Invitation object (I) is an output of the "Prepare Template" tasks and is an input/output to the "Post Invitations" task, and a Single Invitation object (S) is an input/output of the "Fill, Print and

Pack" multiple instance task. We see that in activity-centric models *tasks change object state* and that a *weakly defined association* exists between tasks and objects since tasks are *linked to* object lifecycles.

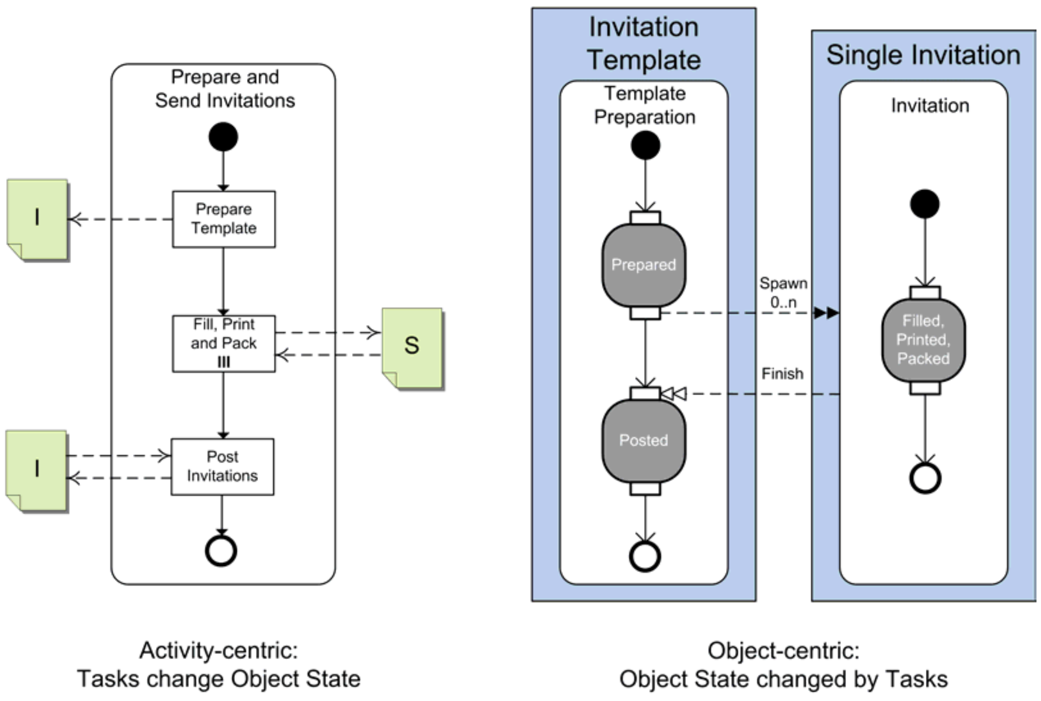


Figure 1.3: Activity-centric and Object-centric Process Representations

This contrasts with an object-centric approach where the tasks in a process *belong to* object lifecycles and *object state is changed by tasks.* In Figure 1.3, tasks that are related to an object are captured by that object. For example, the "Prepare Template" and "Post Invitations" tasks belong to the "Invitation Template" object and the "Fill, Print and Pack" task belongs to the "Single Invitation" object. Tasks are grouped in an object rather than as a sequence in a case, which represents a *strongly defined association* between objects and tasks. This represents an alternative method of designing a process that gives a common criteria for the modularisation of a business process and supporting data. A desirable outcome of this approach is to allow the potential for change. An object and related process logic are located together and tasks are decoupled from inter-object communication, which allows the process logic of an individual object to be changed with minimal side-effects on other objects or on the process at large.

Secondly, there are many issues in process modelling that centre around process flexibility. An advantage of process modelling is the clarity with which control-flow dependencies and relations between tasks in a process are identified and defined. However, merely capturing task dependencies and relations often does not paint the whole picture of control-flow in a process. It is known that during the execution of a process, exceptions to control-flow can occur [10]. Exceptions are caused by the need to deal with work in an unforeseen manner. This is due to situations where case-by-case variations are exposed during process enactment. Case-by-case variations and exceptions commonly occur in domains where an attempt to employ a standard approach to specifying the flow of work is inadequate. These situations are often difficult to appropriately capture using a conventional approach to process modelling without introducing artificial constructs or work-arounds.

In such situations, work is created in an *unplanned* or *on-demand* manner. This causes problems for modelling notations or tools that do not have the capability or functionality to capture the fact that work can be created in these ways, creating the potential to pollute the control-flow in an attempt to support unplanned or on-demand work. Even worse, the correct type of flexibility to use for a particular situation is difficult to define without the ability to handle exceptions of particular types. Thus, a gap still exists in process modelling to allow a satisfactory balance between process structure and the ability to handle unplanned or on-demand work.

In this thesis we investigate how an *object-centric* approach can be used for process modelling. An additional goal is to address existing issues on the topic of process flexibility. Specifically, the problem addressed by this research is to define and evaluate a meta-model for a process modelling language to explicitly address the design of flexible process models.

## 1.3 Research Questions

In light of the above problems, the objective of this research project is to address the following research questions:

- What are the key conceptual differences between object-centric and activity-centric approaches to process modelling?
- What set of concepts are needed in order to model typical business processes from the perspective of business objects and their lifecycles and how do these concepts relate to one another?
- How do object-centric process modelling approaches compare with activity-centric approaches in terms of their ability to capture typical business processes?
- What concepts need to be added into a basic object-centric process modelling approach in order to support highly flexible processes, where process participants can significantly alter the execution of a process?

The first question stems from the observation that the activity-centric approach is presently a more popular choice for contemporary process modelling, as evidenced by the wide-spread acceptance of process modelling notations such as BPMN [66]. Thus, the key concepts of the object-centric approach need to be identified since these object-centric concepts are not as well understood. The second question takes the conceptual investigation into the object-centric approach another step further and motivates the definition of a syntax and a formal semantics of the approach to provide the foundation for object-centric models. The third question highlights the need for an in-depth comparison between the object-centric and activity-centric approaches in order to reveal the strengths, weaknesses and suitability of either approach for process modelling. The final question is inspired by the need to establish if and how an object-centric approach can enhance flexibility in process models.

## 1.4 Research Design

The research design of this project was based on the guidelines of the design science methodology by Hevner *et al* [36]. In accordance with the principles of design science, the project was primarily centered around the development of artifacts. Each artifact contributes in one of two ways to the project. An artifact either provides a contribution to the body of knowledge (including foundation artifacts such as theories, frameworks and methods) or provides a contribution as an application to the appropriate environment (including technological artifacts such as tools and applications).

The artifacts developed during the course of this project were constructed to address the research questions and to provide a clearer understanding of the object-centric process modelling approach through the development of a generic object-centric process modelling language. Three key artifacts were developed: 1) a meta-model for object-centric process models, 2) an object-centric process modelling tool and, 3) a Coloured Petri Net (CPN). The artifacts are not stand-alone and are related to each other in the following ways. The meta-model defines the syntax for the modelling tool and incorporates a set of constructs to capture a set of patterns found in flexible business processes. The meta-model is embodied in the modelling tool which was tested using process models taken from industrial scenarios. A formalisation of the language was defined using a CPN which captures the semantics of the approach and provides the ability to simulate process models exported from the modelling tool.

The meta-model and the CPN are foundation artifacts that adds to the rigor of the research project, whereas the modelling tool is a technological artifact that adds to the relevance. A number of design evaluation methods identified in [36] were used to evaluate the artifacts. The meta-model introduced in Chapter 3 was evaluated using an observational method by implementing and testing the meta-model as the foundation upon which O-C process models can be developed. The modelling tool discussed in Chapters 3 and 4 was evaluated using a structural testing method by implementing and testing real-life models that were contributed by the industry partner using the tool. The CPN in Chapter 5 was evaluated using simulation experiments using data exported from the modelling tool. In Chapter 6 the expressiveness of the meta-model and modelling tool was evaluated by a static analysis method that used the Revised Workflow Control-flow Patterns [83] and Flexibility Patterns [89] as an evaluation framework.

A number of requirements were formulated to address the research questions, which are presented below as six *criteria for a solution.*

1. **The language must be defined in terms of a meta-model.** In this thesis we aim to define an object-centric language to process modelling and investigate applications of the language. The elements of the approach and their associations should be captured as a meta-model to provide the necessary syntax for object-centric process modelling.

2. **The language must be flexible.** The modelling language must have the ability to capture scenarios that experience ad-hoc change and case-by-case variation.

3. **The language must have a graphical modelling notation.** The language must be visual, based on a syntax that is captured as a meta-model that should aid the design of object-centric process models.

4. **The language must be embodied in a modelling tool.** The language must be embodied in a modelling tool to provide a computer-assisted way of constructing and validating flexible object-centric process models.

5. **The language must have a formal grounding.** A formal foundation serves the purpose of removing ambiguity regarding the runtime behaviour of all the modelling elements of the language. On the basis of the formal foundation, the ability to reason about the language and test process models developed using the language to validate behavioural correctness should be possible.

6. **The language must be 'suitable'.** The language must have the ability to support recurring patterns of control-flow in process models and have the ability to support a range of patterns of flexibility.

The criteria for a solution provide the means to evaluate the artifacts. Each artifact was created and evaluated using an incremental development cycle that consisted of three phases: 1) Design, 2) Implementation, and 3) Testing. The phases of the development cycle were repeated until each artifact fulfilled the relevant criteria for a solution. The testing phase of each artifact was completed using data and process models provided by the industrial partner for this project, namely FlowConnect Pty Ltd (formerly Shared Web Services). To evaluate the suitability of the approach for process modelling, pattern-based evaluations of the approach were completed.

## 1.5 Original Contributions

The following is a list of original contributions that are presented in details in this thesis:

- Proposal of a base meta-model for a object-centric process modelling language with constructs that capture the classical workflow control-flow patterns and a transformation procedure from an object-centric model to a YAWL model, and are implemented in a modelling tool.

- An extension to the base model that allows recurrent patterns of flexibility to be captured. Identification and documentation of recurring patterns of flexibility that are included as extensions to the base meta-model, and are implemented in a modelling tool.

- A formalisation of the language using the CPN notation, which enables testing of object-centric model behaviour prior to deployment. The CPN has been designed, implemented and tested using the CPN Tools software.

## 1.6 Thesis Outline

This thesis is organised as follows:

**Chapter 2** introduces related literature in the field of process modelling and flexibility and positions the thesis in terms of the existing body of work from these areas.

**Chapter 3** introduces a base meta-model presented as an Object-Role Model [32] for object-centric process models. The meta-model is used as the basis for a transformation technique to activity-centric process models.

**Chapter 4** presents an extended meta-model that allows flexibility to be handled by capturing unplanned activities in an object-centric process model, while maintaining other desirable properties of process models such as maintaining process control and management.

**Chapter 5** presents a formalisation that captures the operational semantics of object-centric process models using the CPN notation [40].

**Chapter 6** presents an evaluation of the object-centric approach which was undertaken in terms of these two pattern-based frameworks: 1) Revised Workflow Control-flow Patterns [83] and, 2) Taxonomy of Flexibility [89].

**Chapter 7** brings closure to the thesis with an epilogue and a discussion of topics for future work.

# Chapter 2

# Literature Review

This chapter positions the research project by reviewing literature related to process management and object-oriented modelling. In Section 2.1 an evaluation of literature from the field of process management is performed. Section 2.2 compares approaches in the UML for modelling the behaviour of objects and identifies where an extension to state machines can be provided. In Section 2.3, modelling notations that enable process modelling to be combined with object behaviour modelling are investigated. This leads into Section 2.4, where we look at existing approaches towards capturing and managing flexibility in Process-Aware Information Systems and identify an existing gap in previous work done on the topic of enabling flexibility in process models.

## 2.1 Process Modelling

According to Davenport, processes can be understood to mean *"a specific ordering of work activities across time and space, with a beginning, an end, and clearly identified inputs and outputs: a structure for action"* [22]. Process modelling is performed by business analysts using modelling tools to depict the real world processes undertaken by an organisation. These models represent processes from a variety of different scenarios within an organisation that have the means to communicate complex business functions in a format that is intended to be understandable to people. A process model is often used as a foundation to prescribe how things should be done and is essentially a guideline of what the process should look like. The enacted process is determined during the system development lifecycle [81].

Business process modelling is a very broad field of research and practice. This is due to the overwhelming number of modelling notations and languages and that may be used for the purpose of process modelling, each having a different definition in terms of modelling primitives and semantics. Process modelling is an area that has applicability to both software engineering and information systems management since a process model can be used for at least two purposes: 1) As a communication tool, and 2) As an executable software artifact. The list of benefits from modelling a business process includes the possibility for process analysis, process improvement, process automation and process reuse. However we focus on the involvement of software artifacts in a Process-Aware Information System (PAIS), concentrating on process modelling concerns related to process automation (or execution) and as such, take the following definition of PAIS *"a PAIS is a software system that manages and executes operational processes involving people, applications, and/or information sources on the basis of process models"* from [27].

Many notations for business process modelling exist, each presenting different ways of capturing the same thing and having various levels of expressive power. A small subset of these includes general modelling notations that may be applied to modelling outside of business processes such as Petri Nets, to more specialised modelling notations and languages such as Event-driven Process Chains (EPCs) [65], Business Process Modelling Notation (BPMN) [66], Business Process Execution Language (BPEL) [11] and Yet Another Workflow Language (YAWL) [5].

The Business Process Modelling Notation (BPMN) [66] has become a popular mainstream process modelling notation. BPMN makes a distinction between private (internal), abstract (public) and collaboration (global) processes and aims to create a bridge between the design and implementation of processes. Processes in BPMN consist of event, activity and decision flow objects, connected by sequence flow, message flow and association message objects, which are grouped into pools or lanes. BPMN models are not directly executable and must be mapped to an executable language such as YAWL to facilitate implementation of a model captured using BPMN [23].

Commercial workflow management systems (WfMS) that are based upon an activity-centric modelling approach (such as BPMN) are quite common and include systems such as FileNet[1], IBM WebSphere MQ Workflow[2] and Staffware[3]. Activity-centric standardisation efforts also exist such as XPDL [113], which is a language that can capture both the graphical representation and semantics of a BPMN diagram. However, it was noted and demonstrated in [2] that the activity-centric approach is not the most desirable approach for PAIS analysis and design in all circumstances. It was noted that the assumption is commonly made that control-flow within a case is specified in isolation from the remaining cases, resulting in several undesirable outcomes, such as alteration of process specifications to fit a workflow management system, disguising coordination within custom components and potential sequentialisation of tasks that should be executed in parallel.

Some examples of work that has focussed on bridging the gap between conceptual modelling of requirements and deployment of information systems has been contributed by Kueng *et al* [48] and Weber *et al* [106]. Kueng *et al* discuss and demonstrate the application of a methodology to derive the objects or artifacts (the building blocks of an information system) from an analysis of goals, activities and logical dependencies in a problem domain (the requirements). Weber *et al* tackle the same issue from a different perspective by focussing on a structured methodology (a lifecycle consisting of a modelling phase followed by a configuration phase) for mapping the steps (or activities) in process models to IT infrastructure. As has been pointed out by Weber *et al*, the level of guidance provided by the BPM development

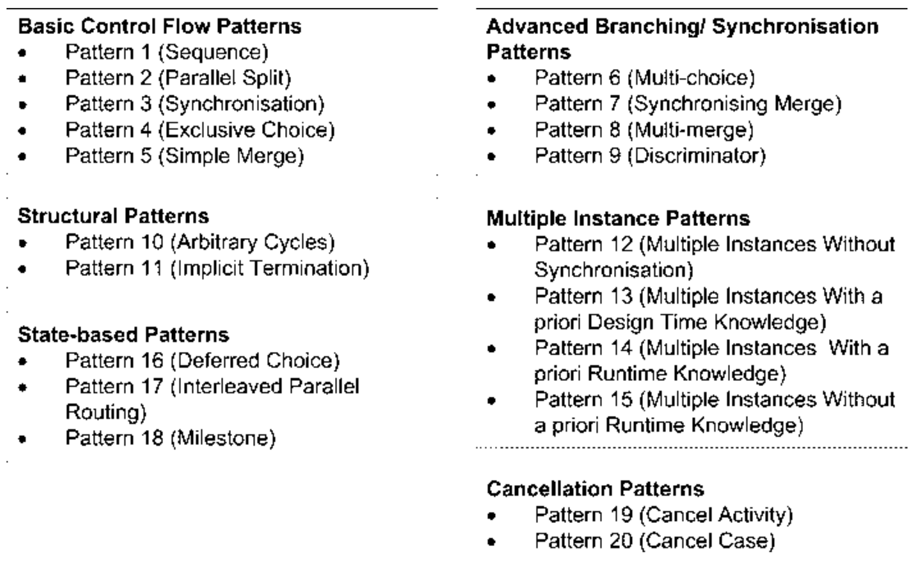
[1]`http://www.filenet.com`
[2]`http://www-306.ibm.com/software/integration/wmqwf/`
[3]`http://www.staffware.com`

lifecycle for process implementation and enactment is usually presented on a fairly abstract level [55].

These works motivate the application of rigorous development methods to implement and enact process models by firstly recognising the importance of model quality and correctness, and secondly, providing a methodology to establish model quality and evaluate model correctness. In other work the business-IT divide is seen as an issue concerned largely with undertaking *business transformation*. Business transformation is the term given to an initiative that seeks to align people, process and technology. An example of a tool that provides decision-support to assist with business transformation named the BT Workblench is introduced and the results of its application to the business transformation is presented by Lee *et al* [54].

To provide a basis upon which meaningful comparisons could be drawn between workflow products, a set of workflow patterns was proposed [6]. A list of the original 20 control-flow patterns, grouped into six classes, is shown in Figure 2.1. The original list of patterns has since been revised and now includes 39 control-flow patterns [83] that can be used to comprehensively evaluate the capabilities of process-aware information systems and process modelling notations with an increasingly specific focus. The intention of the workflow patterns is to provide insights into the comparative level of suitability of a process modelling tool or system, while maintaining independence from any particular vendor, process modelling language or notation.

Initially, patterns were developed that focused upon the control-flow perspective to describe the execution of tasks and their sequence of execution. Workflow patterns that focus upon other perspectives have since been proposed, such as the data perspective [82], resource perspective [86] and exception handling [84]. The wide applicability of the workflow patterns has been demonstrated in a number of research efforts by conducting evaluations of various process modelling languages in terms of the patterns; such as UML Activity Diagrams [112], Business Modelling Language (BML) [110], BPML [4], BPMN [66], BPEL [11] and Open Source Workflow Management Systems [111].



**Basic Control Flow Patterns**
- Pattern 1 (Sequence)
- Pattern 2 (Parallel Split)
- Pattern 3 (Synchronisation)
- Pattern 4 (Exclusive Choice)
- Pattern 5 (Simple Merge)

**Structural Patterns**
- Pattern 10 (Arbitrary Cycles)
- Pattern 11 (Implicit Termination)

**State-based Patterns**
- Pattern 16 (Deferred Choice)
- Pattern 17 (Interleaved Parallel Routing)
- Pattern 18 (Milestone)

**Advanced Branching/ Synchronisation Patterns**
- Pattern 6 (Multi-choice)
- Pattern 7 (Synchronising Merge)
- Pattern 8 (Multi-merge)
- Pattern 9 (Discriminator)

**Multiple Instance Patterns**
- Pattern 12 (Multiple Instances Without Synchronisation)
- Pattern 13 (Multiple Instances With a priori Design Time Knowledge)
- Pattern 14 (Multiple Instances With a priori Runtime Knowledge)
- Pattern 15 (Multiple Instances Without a priori Runtime Knowledge)

**Cancellation Patterns**
- Pattern 19 (Cancel Activity)
- Pattern 20 (Cancel Case)

Figure 2.1: The 20 Original control-flow Patterns

This collection of pattern-based research has provided the possibility to evaluate the capabilities of process modelling languages and notations to form a basis for making meaningful comparisons between the level of functionality that they provide. The results of these evaluations may then enable conclusions to be drawn regarding the comparative expressiveness of a particular system or language. There now exists a substantial number of pattern-based evaluations in the area of activity-centric modelling languages and notations[4]. However from these evaluations it can be seen that the workflow patterns have largely been applied to process-aware information systems and process modelling languages. An observation is that there are relatively few such expressiveness results for object-centric approaches, leading us to consider that the object-centric approach to process modelling is an area worthy of further research.

[4]Workflow pattern self-evaluations of commercial systems are publicly available from `http://www.workflowpatterns.com`

## 2.2 Object Behaviour Modelling

Process models and software models both share notions of structure, behaviour and interaction. As such, it is possible to model processes using the Unified Modelling Language (UML), a collection of modelling notations that are intended to model the structure, behaviour and functions of software systems [18]. The creators of the UML (Booch, Rumbaugh and Jacobson) consider that models are used for several purposes, that models contain semantics and context on several levels of abstraction and that they are representations of the *essential* aspects of a system. As might be expected, the UML is strongly influenced by object-oriented modelling languages, as can be seen by the existence of Class Diagrams and its various types of Collaboration Diagrams in the UML. However, UML also integrates a number of notational elements that go beyond pure object-oriented modelling concepts. In particular, UML Activity Diagrams are intended to support the modelling of business processes from the perspective of activities, focussing on the flow of control and data objects between them. Because object behaviour can be modelled using a variety of notations in the UML, in this section we examine the UML notations that are used to capture object behaviour and interaction, including Activity Diagrams, State Machine Diagrams, Sequence Diagrams and Interaction Diagrams.

An object behaviour model is a static specification of the control-flow behaviour of one or more related classes that shows the effect of control-flow behaviour on object data. In object-oriented (OO) modelling, a class is a conceptual entity that represents either some real or virtual object. A class *encapsulates* the characteristics of an object in terms of the object data and behaviour, and these characteristics may be inherited and specialised from a generalised class. For example, a *Tree* class inherits from the *Plant* class, since a *Tree* is a specialisation of a *Plant*. OO modelling techniques promote *modularity* (i.e. related concepts are grouped together in the same class) and allows the possibility of *reuse*. Since OO focuses upon the definition of data structures and object behaviour, there is an opportunity to address the lack of process modelling support within object behaviour modelling notations.

Attempts have been made to model processes within objects, using Activity Diagrams in the UML [18], for example. UML Activity Diagrams (UML ADs) are a way of capturing the process-related behaviour of one or more objects by focussing on the activities that define object behaviour. Strong points of UML ADs include support for sending and receiving messages at the conceptual level, support for waiting and processing states and an activity decomposition mechanism. Criticisms of UML ADs include a lack of precise semantics and a lack of support for synchronisation patterns [26].

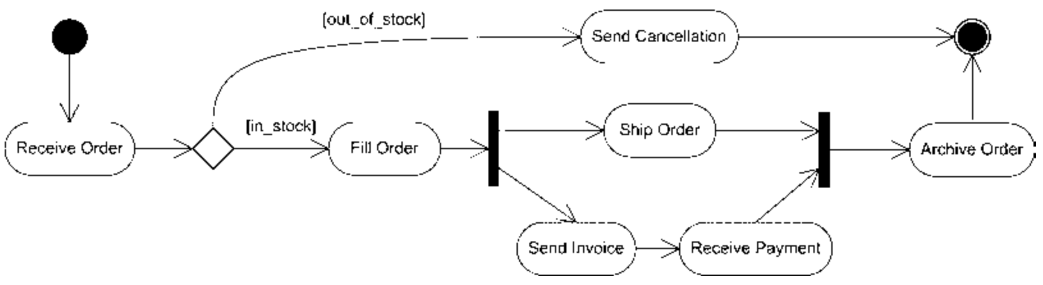


Figure 2.2: Example UML Activity Diagram

A UML AD is used to depict business logic flow and events that cause actions to occur and decisions to be made [71]. UML ADs represent the business and operational workflows of a system, for example the main activities of an order filling system as shown in Figure 2.2. The notions of state and object are not well considered by UML ADs since the focus is on the activities that are performed by a whole system, rather than on the building blocks (and their states) from which the system is assembled. UML ADs offer the notion of an edge called an object flow, but this represents data flow between objects and does not represent control-flow between objects.

It has also been claimed that UML ADs are "not object-oriented", due to their focus on activities within business processes [46]. In addition, it can be considered that UML ADs are not easily integrated with other object behaviour modelling diagrams in the UML such as Sequence Diagrams, State Machine Diagrams and Communication Diagrams, which means that UML ADs can be criticised as having limited value for the purpose of object-oriented modelling. Upon identifing the limitation that UML ADs (along with most other business process modelling languages) have with correctly representing inter-process dependencies, Grossmann *et al* [31] proposed a new *link* construct with four link types (disable, enable, invoke and cancel). Even though we focus on object-centric models rather than activity-centric, from our perspective this is interesting work since objects have the ability to communicate in much the same manner.

Statecharts, as conceived by Harel, are a method for modelling system behaviour [33]. Behaviour in a statechart is captured in terms of the set of states and transitions that transfer control between states. Statecharts were founded on the rationale that conventional state machine notations lead to a potential explosion in the number of states in a model, increasing the amount of difficulty to read and understand a state machine diagram. The statechart notation introduces concepts such as state decompositions that contain other states and/or state machines and support for parallel branches. State decompositions promote modularity and enhance the structure of a state-based behaviour model. However, due to the lack of constructs for inter-process communication and event sequencing, statecharts are an insufficient modelling notation for process modelling.

State Machine Diagrams (based on statecharts) are seen as an appropriate notation for modelling object behaviour due to the conceptual similarities that are shared by state machine models and object behaviour such as modularity, concurrency and hierarchy. Object behaviour is commonly specified using state machines, and as such the State Machine Diagram was adopted by the Unified Modelling Language (UML) for exactly that purpose. A State Machine Diagram allows a designer to model the set of allowable states and transitions that can occur within a model element (e.g. a class) using the state machine graphical notation. An example is shown in Figure 2.3.

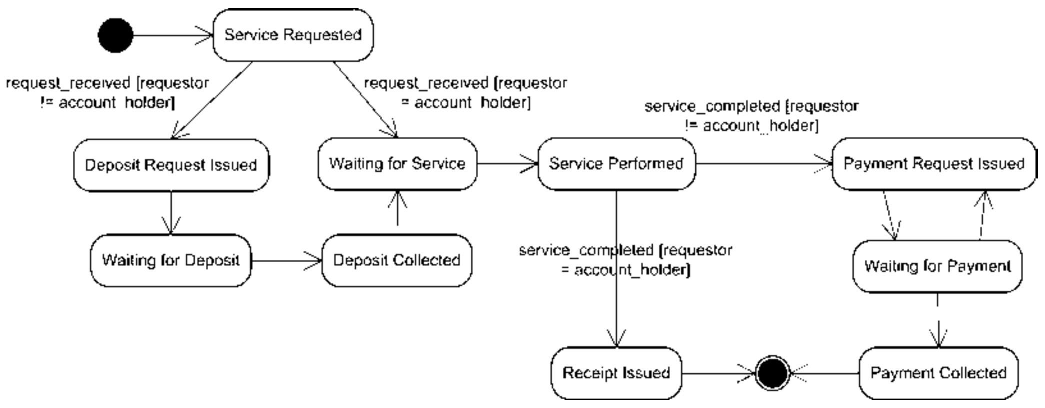

Service Requested
request_received [requestor != account holder]
request_received [requestor = account holder]
Deposit Request Issued
Waiting for Service
Service Performed
service_completed [requestor != account_holder]
Payment Request Issued
Waiting for Deposit
Deposit Collected
service_completed [requestor = account_holder]
Waiting for Payment
Receipt Issued
Payment Collected

Figure 2.3: Example UML State Machine Diagram

Control-flow and communication dependencies between state machines are captured in a State Machine Diagram. However, State Machine Diagrams are distinguished from process modelling approaches, and instead are used for practical graphical modelling of object behaviour. What is not immediately clear in a State Machine Diagram (especially in models of complex systems) is the sequence of control-flow dependencies, or *interactions* between objects. In the UML this gap is filled by additional types of modelling notations that identify and capture different styles of communication between objects such Sequence Diagrams and Communication Diagrams.

A Sequence Diagram is a form of interaction diagram that is reminiscent of a standards proposal by the International Telecommunications Union called Message Sequence Charts [100]. The purpose of a Sequence Diagram is to model the set of message sequences between objects in order to achieve a goal for a specific scenario. Sequence Diagrams specify the sequences of interactions between objects by capturing messages sent and received by each object and the order in which each invocation can occur, shown for example in Figure 2.4. A Sequence Diagram represents the lifecycle of the object that it represents and shows the sequence of the creation of these objects but does not contain details of data manipulation control-flow routing as a result of such communication. In addition, a sequence diagram becomes difficult to follow when modelling multiple branches of messages or handling exceptions. Extensions to Sequence Diagrams can model control-flow, but these are less widely used because it tends to be the case that modelling using these diagrams once again explodes the problem space and leads to the creation of large, unwieldy and overly complex diagrams.

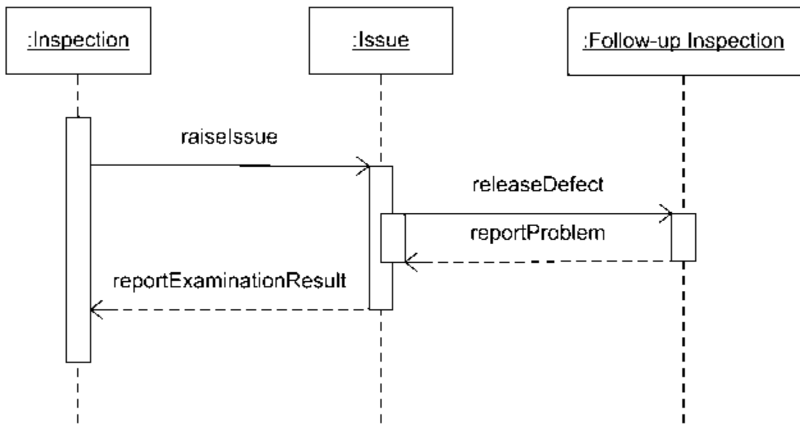


Figure 2.4: Example UML Sequence Diagram

A few approaches have been investigated on the topic of synthesizing object behaviour from scenarios by transformation from sequence diagrams into state machines [17, 35]. These proposals are motivated by the need to show the behaviour of an object (as a state machine) across scenarios (as sequence diagrams) for the purpose of moving from system analysis to system design. In [35], the behaviour of a UML 2.0 State Machine is generated from Sequence Diagram analysis by an algorithmic procedure. In [17], extensions are proposed for both Sequence Diagrams and State Machines in order to allow concurrency, hierarchy and quantification as well as an algorithm that transforms scenarios (a collection of Sequence Diagrams) into a state machine. This provides two sets of object behaviour models. On one hand, a Sequence Diagram captures individual *scenarios* involving many objects whereas a State Machine Diagram captures individual behavioural *constructs* of an object.

The UML is only one possible way of modelling object behaviour. Another example of an object-based framework that has a focus on processes is the Object-Process Methodology (OPM) by Dori [25]. In an OPM model the function, structure and behaviour of a system objects is represented by objects, processes and states. An OPM model concentrates upon defining a system in terms of objects and processes that are connected together with structural or procedural links. A structural link (known as an *association* in UML) connects an object to an object or and object to a process, whereas a procedural link connects a state to a process to capture object behaviour instead of structure. In OPM an object is a container for (and is described by) one or more states, which are the static parts of a system. An object changes state when a process is invoked, which is the dynamic part of a system. OPM recognises that objects are the things that processes act upon and unlike the UML, focuses the emphasis on modelling object function, structure and behaviour using a single modelling perspective.

It is well-known that there exists a large number of object behaviour modelling and specification methods. A comprehensive survey of structured and object behaviour modelling methods was prepared by Wieringa [108]. Weiringa performed an overview of object behaviour modelling methods and classified them in terms of external interaction and internal decomposition. The survey reveals that there is observable convergence between OO modelling techniques and methods, despite their sheer number. For example, a number of methods share the ability to represent system decomposition into objects, object operations and object communications – the main problem is that these methods commonly do the same thing in a different way. A discussion on the use of the UML as a software system specification method is included, which points out that the UML way of decomposing a system is overwhelmingly supported across different OO modelling methods (i.e. decomposition must be represented by a class diagram, component behavior by a statechart, and component communication by Sequence Diagrams or Collaboration Diagrams). Further extensions to the UML are discouraged for the reason of its existing complexity and comprehensiveness, while greater use of formal semantics is encouraged in the definition of modelling techniques (specifically, the UML) to further improve the state-of-the-art of object behaviour modelling.

Finally, an investigation into the reasons why contemporary information systems fail to implement processes on the basis of empirical studies focussed on the implementation phase of the process development life cycle was conducted by Mutschler *et al* [63]. Findings from this work led the authors to indicate that "process orientation" in industry was scarce, leading to an identifiable emerging need for "process-awareness". The study looked at five problem areas facing information systems in the automotive domain. These were: 1) Process evolution, 2) Hard-coded process logic, 3) Complex software customising, 4) Inadequate business functions, and 5) Missing process information. Problems 1 to 3 are fundamental problems that occur during the implementation phase, whereas problems 4 and 5 are related to requirements analysis. The study suggests that organisations commonly do not attempt to upgrade a successfully implemented information system to become process-aware, a phenomenon also noticed by Reijers [78], which raises questions as to the effectiveness of PAIS to handle real-world processes. This motivates the need for PAIS to improve support for flexibility concerns in particular.

## 2.3 Process Modelling meets Object Behaviour Modelling

Object technology is a mainstream approach for the automation of Information Systems in general and PAIS in particular. Mainstream object-oriented analysis and design practices (e.g. those based on UML) are based on concepts of objects whose structure is captured as classes and whose behavior and interactions are captured as state machines, sequence diagrams and similar notations. On the other hand, recent trends have seen an uptake of approaches to Information Systems engineering that treat processes as a central concept throughout the development lifecycle. There are several existing process modelling approaches which have sought to integrate process modelling with object behaviour modelling for many purposes such as improving the design of software systems, reasoning about behavioural correctness, enabling automation or improving process flexibility. In this section we identify and discuss a few notable examples of such integrated systems and modelling notations.

OO design methodologies that use the UML to design and develop Process-Aware Information Systems have been proposed such as OCoN [30, 109]. OCoN is an example of a PAIS modelling notation that identifies the problems associated with process automation when different methods, techniques and tools are used for process modelling and software specification. As a consequence, workflow models are often not in line with the structure of an organisation. OCoN resolves this problem by proposing a modelling notation and an integrated approach to model a process and associated object behaviour in the same model. This approach allows process analysts and designers to obtain process-centric view of an OO model in addition to an automated process model. Weske *et al* [107], Müller *et al* [59] and Johannesson *et al* [41] also present examples of alternative object-centric process modelling approaches.

An object-centric process modelling language can focus on capturing process control-flow behaviour and may not address both data and control-flow perspectives in the same model. An example of such a language is Proclets [2], which is a formal (Petri net-based) model for representing object-based workflows in terms of collections of modules, each of which captures control-flow behaviour (but not data) of a class of objects. Proclets provide a formal basis for reasoning about behavioural correctness in object-based workflows, to identify deadlocks in particular. An area of research that has recently emerged is the *artifact-centric* approach to process modelling [13], based on earlier work reported in [64]. An artifact-centric model explicitly recognises the intertwined relationship between data (objects) and control-flow in a process, and advocates a modularisation of processes around artifacts (which essentially can be considered as business objects).

Previous work in this area has also been reported by Hull *et al* [38] that demonstrates an approach to workflow modelling by combining declarative workflow and process artifacts. Hull also outlined four dimensions for future research into artifact-centric business process models, which are: 1) identification of business artifacts, 2) artifact lifecycle modelling techniques, 3) service and task modelling and, 4) definition of associations linking artifacts to services [37]. The four dimensions are meant to emphasis the separation of the main concerns of artifact-centric models while allowing the possibility to cater for both procedural and declarative process models, thus enabling the approach to capture both rigid and flexible structures.

Artifact-Centered Operational Modelling (ACOM) is a methodology introduced by systems designers at IBM to develop process-based models and systems based on artifact lifecycles [12]. ACOM focuses on defining the operational goals of an organisation, such as "processing a purchase order", by linking operational goals to artifacts. An artifact lifecycle is specified in terms of artifact tasks. An artifact task involves bringing together information from one or more artifacts so that the task can be completed. Completion of a task results in a change of the business state, although artifact tasks have no knowledge of the business state or indeed of the tasks that precede or follow the tasks. Instead, artifacts take care of coordination between tasks and contain the business state. Model-driven Business Transformation (MDBT) [49] is a method of transforming an artifact-centric model into an executable system by mapping business operation models into a solution design model. Combining the ACOM approach with the MDBT method shows how the idea of involving artifacts with process models provides a central step towards realising the vision of Model-Driven Architectures.

Other research proposals such as $\text{WASA}_2$ [103] and $\text{TriGS}_{\mathit{flow}}$ [42] have presented methods how object-centric workflow systems may handle dynamic flexibility and perform database transactions. It can be seen from these proposals that object-centric systems may inherit limitations from being tied to implementation technologies, such as CORBA [67] in the case of $\text{WASA}_2$ and the Gemstone OODBMS[5] in the case of $\text{TriGS}_{\mathit{flow}}$. Additionally, it should be noted that object-centric systems are not found in the commercial marketplace as commonly as activity-centric systems. Examples of these

[5]`http://www.gemstone.com`

commercial systems include Visuera[6], Bonita[7] and Metocube[8]. Another integrated approach is the Business State Machines model supported by IBM Websphere Process Server [39], which relies on a paradigm based on communicating state machines. It should also be noted that an integrated approach is not to be confused with types of process management paradigms that consist of an API that happens to be encoded in a particular object-oriented language (e.g. jBPM[9]).

Kim *et al* [45] has proposed a design methodology for process models from a slightly different perspective. The approach by Kim links UML diagrams to phases of the process development lifecycle to produce a schema as output at the conclusion of the lifecycle. There exists a possibility to extend this methodology to allow process analysts and designers to create an activity-centric view of an object-centric schema by means of providing transformations between object-centric and activity-centric models.

There are many existing proposals that have sought to address the observable mismatch between object models and process models by way of *transforming* one approach to the other. Here we look at a few notable proposals made on this topic. OO modelling approaches have been used as the base model for process mapping. An architecture for mapping between OO and activity-centric process modelling approaches has been proposed by Snoeck *et al* [96]. Object associations and business rules are captured using object-relationship diagrams and an object-event table models the behaviour of domain objects.

Along much the same lines, a proposal to unify the views of state and behaviour has been proposed by Wagner that formalises the associations between agents, objects and relationships in a process model [104]. Wagner introduces a modelling notation that combines UML Activity Diagrams with statecharts to achieve the ability to capture both state and behaviour in a single model. Aspects that appear to be missing from this proposal

[6]`www.visuera.com`
[7]`bonita.objectweb.org`
[8]`www.metocube.com`
[9]`www.jboss.org/jbossjbpm/`

includes a consideration of the various kinds of control-flow splits and joins between objects. This control-flow information is necessary for process model specifications and this information should be derived directly from an object-oriented analysis.

Küster *et al* [52] define a transformation from object behaviour models to process models. In this proposal object behaviour models (called object life-cycles) are represented as state machines and process models are represented as UML activity diagrams. In the meta-model considered in Küster *et al*, synchronisation dependencies are captured as *synchronisation events* which are akin to synchronous message exchanges between one object of one type and one object of another type. However, Mohan *et al* [57] suggest that the overheads involved in the deployment of e-commerce platforms forces companies to adjust their business processes to conform with the system rather than modifying the system to match the preferred business processes and propose a tool called FlexFlow to model applications using statecharts. This is a motivation for a state machine-based (object-based) approach to model, enact and control business processes within applications.

The Mentor project proposed an approach to combine state charts and activity charts to develop workflow specifications [62]. Mentor is an architecture that is based upon the execution of state and activity charts. This project presented an algorithm to perform transformations between centralised and distributed representations of a workflow by converting specifications from other languages into state and activity charts then partitioning activities in the workflow into sub-workflows, thereby "clustering" related activities together. Such a method is of interest when considering how an activity-centric (or globalised) model could possibly be converted into an equivalent object-centric (or localised) model.

Kumaran *et al* [50] reports on an approach to link business entities with activity-centric process models for the purposes of providing an alternative method of process automation by connecting business processes (control-flow) to business entities (data). This research suggests that the following properties should be exhibited by information entities (or objects): 1) Entities are self-describing, 2) Entities are organised into non-overlapping process partitions, and 3) Entities are decoupled to allow independent evolution.

Loos and Allweyer [56] have shown how to link EPC with several UML modelling notations including state machines, activity diagrams, use cases, sequence and collaboration diagrams. This work provided an early example of how business process models could be used as a starting point for the specification of object behaviour, by demonstrating how the high-level control-flow captured by EPC can be merged with systems modelling notations that more closely resemble an implementation. Several reports have been made on linking Petri nets to either BPM or object-oriented models, in which process structures are commonly derived in terms of Petri nets and then modularised into individual sub-nets/pages or objects [53, 58, 20]. These proposals show the applicability of Petri nets in the domains of BPM and OO modelling.

In Figure 2.5 we categorise several modelling tools from research and industry mentioned in this chapter into a spectrum of modelling approaches, represented by tools that can be classified along a spectrum of modelling approaches that has two extremes: activity-centric and object-centric. These two approaches have fundamental differences towards process decomposition. At one end of the spectrum is the activity-centric approach which has an emphasis on control-flow aspects of a process, meaning that functions and control-flow between functions are used as the main driver for decomposition of a process. Some tools that represent this approach include YAWL, BPMN and TIBCO Business Studio. While use of the activity-centric approach is currently more popular (i.e. usage indicators of BPMN are reporting widespread uptake), it has been noted that this modelling approach tends to have the effect of "flattening" a process since they are unable to account for the mix of perspectives that exist in a real process [2].

At the other end of the spectrum is the object-centric approach where a process is decomposed into a hierarchy of modules, according to the data or artifacts that belong to the process. Modularisation has been identified as a decomposition technique that improves flexibility and comprehensibil-

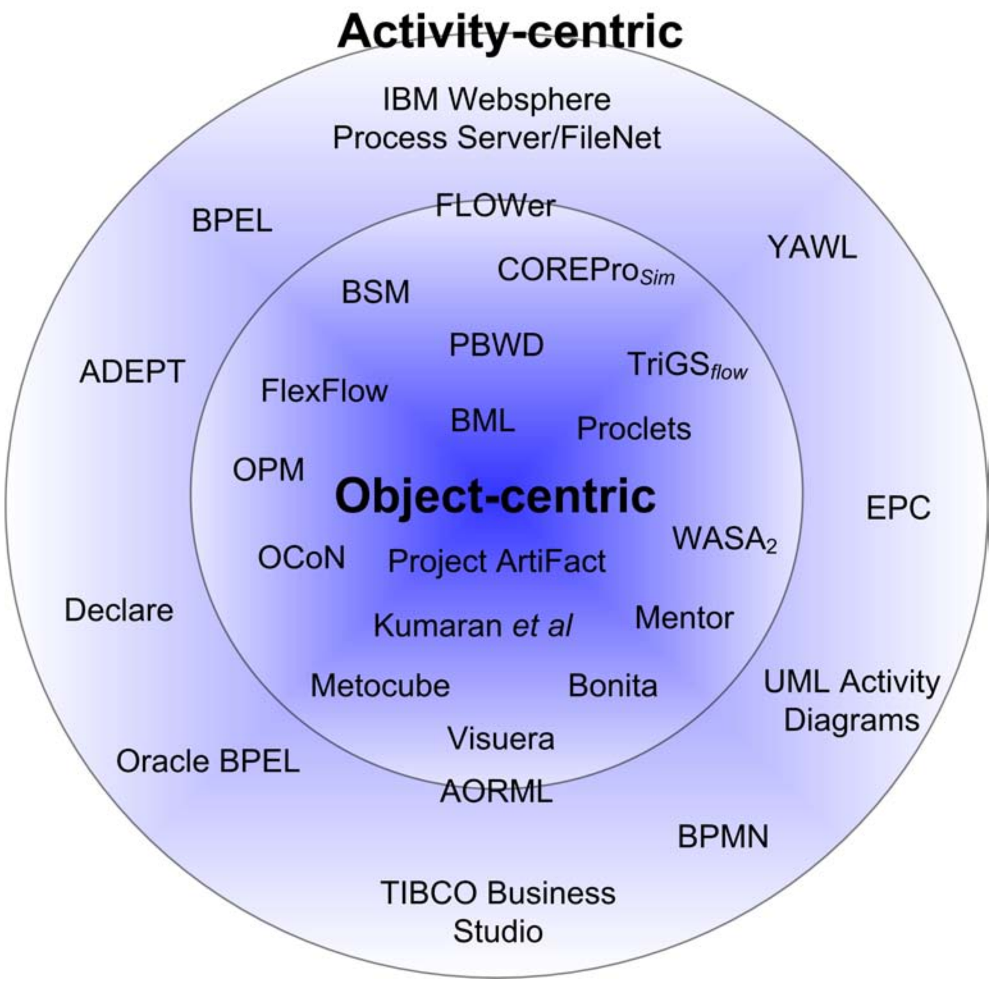


Figure 2.5: Business Process Modelling Spectrum

ity while shortening development time through module reuse [68]. Tools that represent this approach include BML [110], BSM [39], OCoN [109] and COREPRO$_{Sim}$ [60]. There are also some tools that are found to exist in the middle of the spectrum, such as FLOWer [8] and the Agent-Object-Relationship Modeling Language (AORML) [104]. These tools do not use a fully activity-centric approach, or a totally object-centric approach, but rather tend to be based on other modelling paradigms such as case handling, which combines aspects from both activity-centric and object-centric modelling approaches. The spectrum of modelling approaches shows us that a variety of approaches exist that are essentially used for the same purpose, but allows us to point out that the fundamental drivers of each approach are quite different.

Following a review of related literature in the area of process modelling we can identify several current gaps that exist in the body of knowledge. Overall, it can be seen that an object-centric approach to process modelling has not received as much attention as the more conventional activity-centric approach. However, there are several proposals that have identified the potential role that objects might play in improving the state-of-the-art in process modelling. This has also been recognised by industry to a certain extent with the appearance of commercial tools that capture and implement processes on the basis of object lifecycle modelling and execution. There exists the opportunity to extend this body of work to integrate concepts found in the software engineering Design Patterns [28] concerned with the creation of sub-processes (or objects), along with some more advanced synchronisation constructs that are not found in the revised workflow control-flow patterns [83] to do with synchronisation after having received a threshold number (maybe also in combination with particular object types) of object lifecycle completions. Literature in the area of process flexibility is reviewed in the next section to identify some gaps in research with respect to handling flexibility in object-centric process models.

## 2.4 Process Flexibility

It has been recognised in the literature for some time that Process-Aware Information Systems, such as traditional Workflow Management Systems, often have difficulties supporting dynamic business processes because they rely on modelling paradigms that tend to impose a given execution order between activities and decision points. This fact has been discussed in the literature for some time which has resulted in many proposals for flexible workflow languages and tools (e.g. [16, 34, 107, 69, 105, 21]).

Other OO or object-based process modelling approaches have been proposed by Küster *et al* [52] and Wirtz *et al* [109]. However, these latter proposals are not motivated specifically by flexibility requirements. For instance, the work of Küster *et al* is instead motivated by compliance management. An alternative paradigm to OO process modelling is *case handling* [8], a paradigm for business process support where the primary focus is on the data supporting a system rather than purely on capturing control-flow behaviour. Snowdon *et al* observes that achieving flexibility is tightly linked to relaxing control and suggests that shifting focus away from pure control-flow can lead to less restrictive systems [97].

Bider and Khomyakov have motivated the use of a state-based approach to achieve process flexibility [15, 14]. Related states are grouped together with relevant laws (business rules) and connectors (external dependencies) to form an object. The argument given for a state-based approach depends on the ability to abstract away from ordering of activities to focus on the higher-level organisational goals. A goal is achieved when the final state(s) of an object are reached, otherwise progress towards achieving a goal is still being made. The focus on goals and states in this approach represents a departure from how process models are typically designed.

From the literature it can be seen that there is a significant amount of existing research related to *flexible process management*. Research in this field has focused on dealing with runtime deviations with respect to the expected execution of a process model (*dynamic change*). A proposal consisting of five criteria for characterizing dynamic change [80] shed more light into the shortcomings of conventional process management systems with respect to flexible execution, and enabled comparative evaluation of the change-handling capabilities of a number of process management systems. Weber *et al* [105] built on top of this work by defining 17 change patterns.

### 2.4.1 Observations on Flexibility

Work by Klingemann observed that true benefits are gained from flexibility in process models when flexibility is controlled [47], an observation we agree with. To achieve flexibility in process models Klingemann classified three flexible element types; alternative activities, non-vital activities and optional execution order. There is room remaining on top of the work of Klingemann to extend his classification to cater for additional types of process intervention that can deal with *unplanned* activities. In addition to catering for different types of flexible elements, there should be alignment between computerised and real-world processes. This position is shared by work done on the ADEPT$_{flex}$ [76] project and Borch *et al* [19].

Snowdon *et al* observes that achieving flexibility is tightly linked to relaxing control and suggests that shifting focus away from pure control-flow can lead to less restrictive systems [97]. In work done by Sadiq *et al* [87], it was observed that several dimensions of change may affect executing instances of a process, namely dynamism (the ability to evolve), adaptability (the ability to react) and flexibility (the ability to defer process specification until runtime). To handle these dimensions of change, a framework called "pockets of flexibility" was proposed to define flexible workflow specifications that can be tailored to runtime circumstances. Recent work by Seidel *et al* has proposed a conceptual framework for information retrieval to support pockets of flexibility and presents how this approach can be practically applied within highly creative, adhoc scenarios such as movie production [93].

In the field of workflow escalation, Georgakopoulos et al [29] outline an approach to support dynamic changes in workflows in *emergent* situations (e.g. for rescue operations during natural disasters). Their focus is on enabling decision makers to escalate tasks at runtime by changing the course of the workflow execution as required, while retaining some level of control. These concepts promote a balance between structure and flexibility. In contrast, our work focuses on capturing runtime variability of workflows at design-time, instead of escalation. In addition, neither the approach proposed by Sadiq *et al* or Georgakopoulos *et al* give any indication of the role that business objects might play in their proposals.

### 2.4.2 Taxonomy of Process Flexibility

A Taxonomy of Flexibility proposed by Schonenberg *et al* [89] identifies and defines four types of flexibility in processes. These types are Flexibility by Design, Flexibility by Change, Flexibility by Deviation and Flexibility by Underspecification. An explanation of each type of flexibility is listed below:

- **Flexibility by Design:** for handling anticipated changes in the operating environment, where supporting strategies can be defined at design-time. Aspects of a notation or system that are related to flexibility by design include the ability to fully support parallelism, choice, iteration, interleaving, multiple instances or cancellation.

- **Flexibility by Deviation:** for handling occasional unforeseen behaviour, where differences with the expected behaviour are minimal. This includes the ability to perform actions on tasks such as Undo, Redo, Skip, Create additional instance or Invoke task.

- **Flexibility by Underspecification:** for handling anticipated changes in the operating environment, where strategies cannot be defined at design-time, because the final strategy is not known in advance or is not generally applicable. This may be achieved in several different ways, including late binding (at design-time), late modelling (at runtime), before placeholder execution, at placeholder execution, with static realisation or with dynamic realisation.

- **Flexibility by Change:** either for handling occasional unforeseen behaviour, where differences require process adaptations, or for handling permanent unforeseen behaviour. Flexibility by change can be further divided into two sub-classifications: momentary change (at the instance level) or evolutionary change (at the type level).

As shown in Figure 2.6, the types of flexibility in this taxonomy can be characterised by looking at two key process model attributes. Firstly, a process model specification is either partial or full, and secondly, flexibility can be applied at either design-time or runtime. Characterising process models in this way allows us to see that each type of flexibility covers a different cross-section of process models. For example, a partially specified process model at design-time requires Flexibility by Underspecification (late binding) whereas a fully specified process model requires Flexibility by Design.

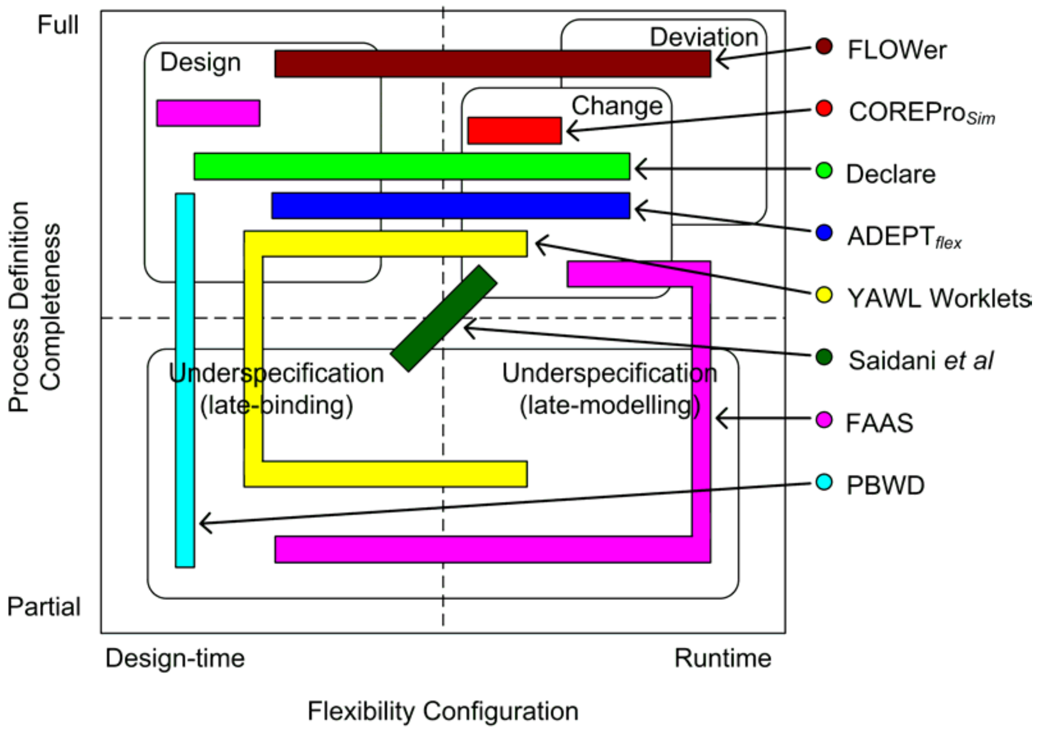


Figure 2.6: Spectrum of Flexibility Types, inspired by [89]

This taxonomy reveals a characteristic that, in general, a particular process modelling notation provides a good level of support for flexibility by design and one other type of flexibility. A potential field for investigation is to evaluate whether a combined process modelling and object behaviour modelling approach can achieve a greater level of support for the flexibility patterns than has otherwise been observed in other process modelling notations. In Figure 2.6, a set of flexible modelling languages have been overlaid to the taxonomy to indicate which types of flexibility they support. The list of modelling languages is DECLARE, COREPRO$_{Sim}$, ADEPT$_{flex}$, YAWL

Worklets, Flexibility as a Service (FAAS), Product-Based Workflow Design (PBWD), FLOWer and work by Saidani *et al.*

### DECLARE

DECLARE [70] is an example of a constraint-based workflow modelling tool that describes loosely structured processes, which is achieved in DECLARE by using a declarative modelling approach. This approach to workflow modelling grants a process designer the ability to focus on the 'what' rather than the 'how'. The strength of this approach is that model constraints can be added or relaxed (made optional) where needed. This allows a process expressed with DECLARE to capture several types of flexibility, in particular flexibility by deviation and flexibility by change. Support for flexibility by deviation is achieved by allowing constraints to be specified as optional constraints, thereby enabling control over constraint violation. Support for flexibility by change is achieved by allowing both instance and type level changes at any specified time during process execution. However, DECLARE is not suitable for modelling all types of processes, large processes and highly constrained processes in particular, due to the high number of constraints which results in a model that is difficult to understand. Performance issues have also been identified with this approach.

### COREPRO$_{Sim}$

COREPRO$_{Sim}$ [60] is an example of a process modelling tool that provides a way of modelling a data-driven approach for process specification. The approach corresponds to flexibility by change. COREPRO$_{Sim}$ achieves flexibility by change by translating information regarding the structure of a product to an adaptation of the process structure. The COREPRO$_{Sim}$ process model can then adapt to changes in the product structure, either static changes at design-time or on-the-fly changes that can occur at runtime, while restricting process adaptations that would lead to violating soundness properties such as deadlock. This approach, while very interesting, has multiplicity constraint limitations between objects. In particular, the notion of an "external state transition" between object lifecycles can only be applied to circumstances where there exists one parent object instance and one child object instance (representing a *1..1* relation), resulting in large product structure models to capture every dependency between each object.

### ADEPT$_{flex}$

ADEPT$_{flex}$ [76, 77] is an example of an adaptive workflow language that allows runtime change to both running instances and the workflow schema to omit, insert or change activities. ADEPT supports flexibility by design and flexibility by change. Elements of flexibility by design that are supported include parallelism, choice and iteration, while flexibility by change is supported by allowing change to occur at any time to an executing instance. However, all tasks in ADEPT are mandatory and there is no possibility to specify optional *tasks* (as opposed to specifying optional *task parameters*). Also, additional task instances cannot be created. While presenting a variety of flexibility options to workflow instances, a workflow schema created in ADEPT still represents a prescriptive way of working.

### YAWL Worklets

YAWL Worklets [9] are an approach that allows dynamic evolution of workflow specifications, through a mechanism that handles exceptions as they occur in a running YAWL process model. A worklet is a sub-process that may be dynamically called during an instance of a process to provide a method to deviate from the planned way to complete a process. Worklets have the advantage that the original process specification does not need to be redesigned in order to cope with unforeseen circumstances. This approach enables a YAWL specification to support some flexibility by underspecification patterns since it allows late binding, late modelling and dynamic placeholding of the nodes from where a worklet can be enacted. However, a YAWL Worklet behaves somewhat like a black-box because it cannot communicate during its lifecycle with the parent process that started it. In addition, after starting a Worklet the parent process waits at the same point for it to finish in the same manner as a task. These represent control-flow limitations that should be made possible to overcome.

### FAAS

The FAAS proposal [1] is a novel structured approach inspired by the taxonomy of flexibility that enables a process designer to combine the flexibility aspects of three process modelling approaches, namely YAWL [5], Declare [70] and YAWL Worklets [9]. The reasoning behind combining these approaches is gain the relative benefits offered by each approach. At the implementation level, this approach allows a process designer to take advantage of accessing the flexible aspects of these different technologies by arbitrarily nesting them. Thus, a range of flexibility types can be claimed to be supported. An interesting approach worth investigating would be to evaluate whether a similar spectrum of flexibility types could be supported without the need for combining different technologies since each type of technology has its own approach to workflow specification, which steepens the complexity of process design due to the need to know (and combine) multiple languages rather than a single language.

### PBWD

PBWD [79, 101] is an approach to specify a process model in accordance with the structure of a product (its “bill-of-material”). A product in this context may be either a physical good (such as a car) or a service (such as awarding unemployment benefits). PBWD is a revolutionary approach to process modelling, since a cleansheet approach to process design, rather than an evolutionary approach. Flexibility in the design of a product data model in PBWD is achieved by leaving the exact path of execution of a process unspecified until runtime, meaning that PBWD supports elements of flexibility by design and flexibility by underspecification. However, PBWD focuses on inferring the high-level structure of a business process, and not its detailed behaviour. In particular, the PBWD approach does not consider the issue of describing the behaviour of product lifecycles in an executable manner (e.g.

as state machines), and automatically generating business process models from such lifecycles.

### FLOWer

FLOWer [8] is a tool provided by a company called Pallas-Athena. FLOWer is an example of a *case handling* system, where the path of process execution is loosely defined. In case handling the main concern is instead shared between appropriate resource allocation to enabled tasks and obtaining sufficient data to complete a case (also known as a process instance). The result is that the focus of FLOWer process models is upon what *can* be done rather than what *should* be done to complete a case. In terms of the flexibility patterns, case handling is a very flexible approach due to its support for elements of both flexibility by design and flexibility by deviation (skip, undo, redo). However, FLOWer does not support flexibility by underspecification which means that *unplanned* tasks cannot be brought into the case.

### Saidani *et al*

Issues related to delegation and role-based constraints that must be handled by flexible process models have been investigated by Saidani and Nurcan [88]. This work gives insight into delegation (flexibility by underspecification) and role-based constraints (flexibility by change). To support such types of flexibility, a meta-model is proposed that addresses the complexity involved in balancing concerns on three levels: 1) control-flow flexibility, 2) resources involved in a process and 3) constraints that must at times be enforced by a process. Particular recognition is given to delegation, since the act of delegating (while often a beneficial flexible tool) can cause more problems than desired. To alleviate these potential problems, limitations are proposed on delegation that aim to restrict the ability of a role to delegate within certain control-flow and role-based constraints.

As we have witnessed from the literature presented in this section, enabling flexibility is a current concern in process modelling. There exists several different proposals that have analysed and classified types of flexibility that affect PAIS modelling and execution. The intent of the thesis is to demonstrate support for multiple aspects of flexibility rather than provide support for any particular flexibility type. From the literature we can see different approaches tend to strongly address certain groups of patterns of flexibility while only weakly addressing others. In this work we seek to support several different types of flexibility from different categories and achieve this by using one process modelling approach.

## 2.5 Summary and Discussion

As introduced and discussed in Chapter 1, the business-IT divide is an ongoing problem facing the development and implementation of PAIS. Over time, several approaches have contributed to "closing the gap", resulting in the present-day situation where the discipline of Business Process Management and Business Process Modelling is positioned to provide the next wave of (process-aware) information systems with the capability of improving the alignment between business and IT. In this context, it is interesting to imagine the possibilities that may emerge as a result of combining process models (how an organisation does things) with object models (the building blocks of an information system).

In this chapter, a review of the literature that focussed on object-centric process modelling approaches and process flexibility was presented. We have identified object-centric process modelling as a worthy topic of further research for the following reasons: 1) There appears to be benefits to be gained from further work into object-centric process modelling, notably in the areas of ad-hoc multiple instance creation and synchronisation, and 2) Flexibility in process models could benefit from taking on an object-centric approach.

In the next chapter, we take the first step of this research project by introducing a meta-model for object-centric process modelling to provide the foundation for investigating the research problems identified in Chapter 1.

# Chapter 3

# A Generic Object-centric Process Modelling Meta-Model

In this chapter we introduce the abstract syntax of a generic object-centric language for process modelling. The syntax is defined by a meta-model defined using the Object Role Modelling (ORM) [32] notation. A proposed notation for the object-centric modelling language is presented as a way of capturing design-time information in a process model using the OO paradigm. An application of the notation is demonstrated using an example taken from an industrial scenario of a maintenance process for gas pipelines. This serves to illustrate how control-flow behaviour in a process model can be captured by objects and their associations. We then investigate the relationship between this object-centric meta-model and a mainstream activity-centric process modeling language, namely YAWL, by defining a transformation between the former and the latter. As a result of this investigation, we are able to demonstrate how object-centric meta-models can be transformed into activity-centric ones. This chapter is based on work published in [73] and [74].

## 3.1 Object Behaviour Meta-Model

Object-oriented modelling is a paradigm that allows one to document the structural, behavioural and creational aspects of a system in an object-oriented way. In this section we will focus upon extracting the salient aspects of an object behaviour meta-model that allows a model designer to capture a process using objects. To capture the syntax of object models we present a meta-model inspired by several concepts that are found in the FlowConnect system developed by FlowConnect Pty Ltd [94], the industry partner for this project. FlowConnect is a system that supports the development of software applications based directly on executable object behaviour models, which provides us with an attractive source meta-model for two reasons.

Firstly, the FlowConnect system seamlessly integrates concepts from UML state diagrams with concepts from UML sequence diagrams, allowing us to capture both intra-object and inter-object behaviour in the same model. It is advantageous to capture intra-object and inter-object behaviour in the same model in order to get a better perspective of object control-flow relations, state machine behaviour and dependencies. Secondly, the FlowConnect-based meta-model is a representative of other object-oriented meta-models that are also founded on a state-based paradigm (e.g. Proclets [2], Merode [96], OCoN [109]), thus it can be claimed that the results presented here may be adapted to other meta-models. The meta-model is presented as an ORM in Figure 3.1.

We now describe in details the characteristics of each element and the relationships between elements in the meta-model. At the highest level in the meta-model an **object model** is a container for all classes in an object-centric model. The object model contains one or more **classes** that contain one or more **state machines**. We define a class as a cluster of state machines that share some common context. An example of shared context are states that are all predominantly involved in a *Primary Inspection*, which are grouped together to allow the common concepts contained within a *Primary Inspection* class to take shape.

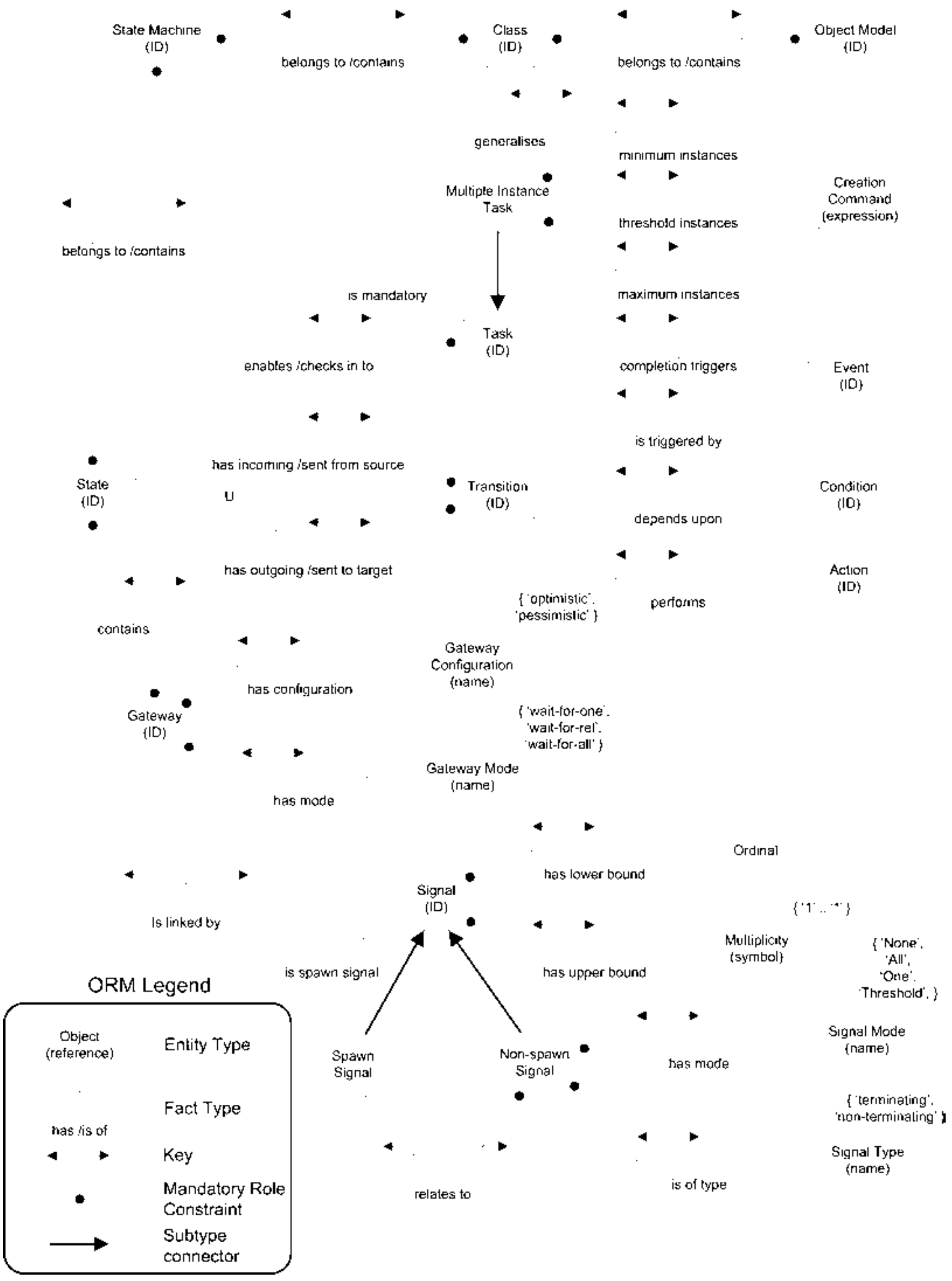


Figure 3.1: Object Behaviour Meta-Model in ORM

A state machine contains an **initial state**, a **final state**, and one or more **states**. The initial and final states are pseudo-states that explicitly indicate where the beginning and end of a state machine lifecycle are found. A state machine has exactly one initial state and exactly one final state. The initial state is the only state in a state machine that has no incoming transition. Likewise, the final state is the only state that has no outgoing transition. When an instance of a state machine is created, the execution begins in the initial state. When a final state is exited, the state machine instance that the final state belongs to is terminated. Other states are used to define each step in a state machine lifecycle. A state represents a moment in a state machine lifecycle that can be distinguished and given a unique name, for example, a series of states in the state machine of a primary inspection are *Report Distributed*, *Data Loaded* or *Structural Strength Tested*.

A **transition** connects the output of a state (the source state) to the input of another state (the target state) in the same state machine. Transitions may have an optional Event-Condition-Action (ECA) rule. The occurrence of an **event** will cause the transition labeled by that event to be performed. A transition can have a **condition** associated with it that must be satisfied before the transition can occur. When a transition is performed it may also execute an **action**.

Each state contains three sub-states; a **pre-gateway**, **processing sub-state** and **post-gateway**, as shown in Figure 3.2. The processing sub-state is where zero or more atomic task instances are started and completed. The pre- and post-gateways are the entry and exit points of the processing sub-state respectively. The pre-gateway is entered after a transition has been performed to the state that it belongs to. The processing sub-state is entered after the pre-gateway is entered. Lastly, the post-gateway is entered after the processing sub-state has exited.

In our generic meta-model the processing sub-state is a place from which zero or more atomic **tasks** are enabled. After control-flow enters this sub-state the tasks that are enabled by that state may then be started, executed and completed. A task can be a mandatory task, which means that it is

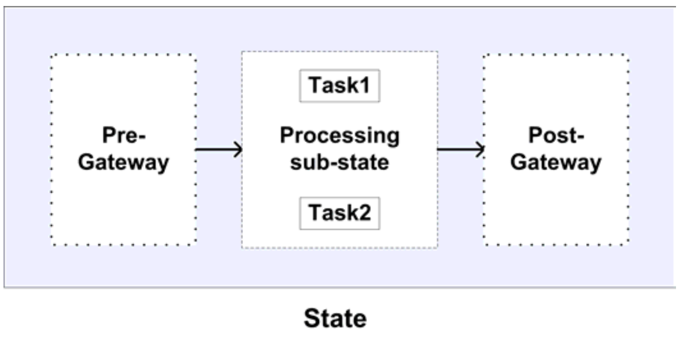


Figure 3.2: Pre-Gateway, Processing Sub-State and Post-Gateway

necessary for the task to have been completed before the sub-state can be exited. Since a state can have more than one task, all mandatory tasks within a state must be completed before the sub-state can be exited.

Conversely, tasks that are not mandatory are optional tasks. Optional tasks are not required to be completed before exiting the processing sub-state. If the sub-state is exited and optional tasks have not been completed, then these optional tasks remain active and can still be completed, but the completion of an optional task has no effect on the control-flow. A task can be a multiple instance task, which means that there are a minimum number of task instances that must be completed, a threshold number of instances that may be enough to consider the multiple instance task is completed and a maximum number of instances of that task which may be completed. A multiple instance task can also be mandatory, meaning that all instances of the task that were started must be completed before the processing sub-state can be exited.

The pre-gateway and post-gateway of a state both have an input part and an output part. For the purpose of enabling inter-object communication and synchronisation between state machines, a gateway can send signals to other state gateways and/or receive (wait for) signals from other state gateways. The input part is the target of zero-to-many incoming signals and the output part is the source of zero-to-many outgoing signals. The input and output parts of a gateway are given an ordering. The ordering depends upon the **gateway configuration**, which is either *optimistic* or *pessimistic*.

As shown in Figure 3.3, an *optimistic* gateway configuration has the output part before the input part, meaning that it sends signals before waiting for signals whereas a *pessimistic* gateway configuration has the input part before the output part, meaning that it waits to receive before sending.

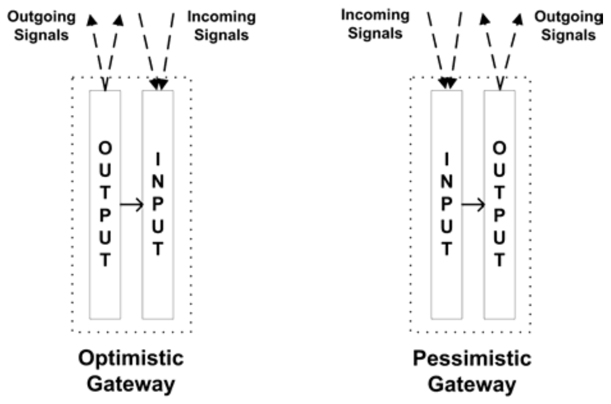


Figure 3.3: Optimistic and Pessimistic Gateways

These gateway configuration options follow a similar approach to what was reported by work on the OR-Join by Wynn *et al* [114] although our definition of optimism and pessimism is slightly different since in this case a gateway is more than a join; it is a combination of a join and a split with a configuration that specifies the order of the join and the split. To clarify, there are four possible optimistic and pessimistic gateway configurations for a state which are presented in Figure 3.4.

The input part of a pre- or post-gateway has a **gateway mode** that is used to specify the control-flow blocking behaviour of a gateway. The gateway modes are *wait-for-one*, *wait-for-rel* and *wait-for-all*. The *wait-for-one* mode specifies that the gateway should release control-flow after receiving a single signal from any source. The *wait-for-rel* mode specifies that the gateway should wait for all signals from any relationship between two state machines. The *wait-for-rel* mode is used when a gateway receives signals from two or more state machine types and control-flow is released when all instances of any state machine type have sent a signal. The *wait-for-all* mode specifies that the gateway will wait for all signals that are expected

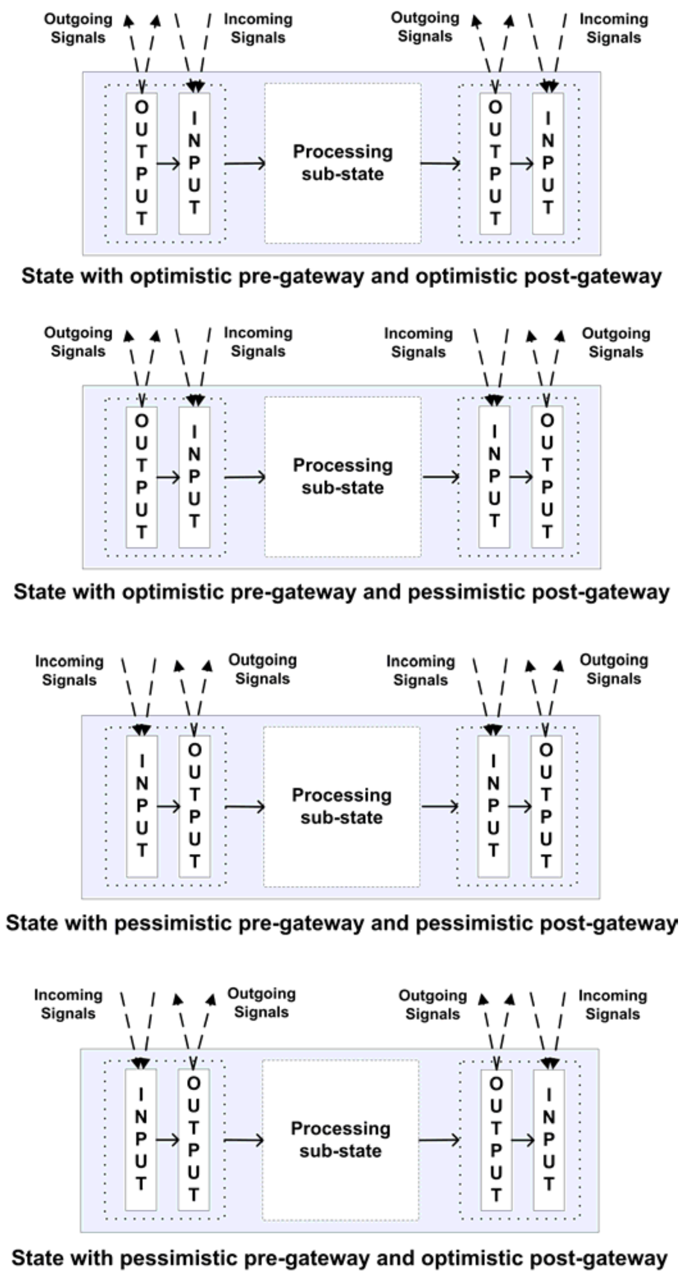

Outgoing Signals
Incoming Signals
OUTPUT
INPUT
Processing sub-state
State with optimistic pre-gateway and optimistic post-gateway
State with optimistic pre-gateway and pessimistic post-gateway
State with pessimistic pre-gateway and pessimistic post-gateway
State with pessimistic pre-gateway and optimistic post-gateway

Figure 3.4: Optimistic and Pessimistic Gateway Configurations

from all sources that are connected to that gateway. The *wait-for-all* mode has a special caveat, since if a *wait-for-all* gateway is waiting for zero signals (no state machine instances that send a signal to the gateway were created), control-flow will be released even though no signals were received.

In the meta-model there are three different signal types. A **signal** establishes a one-way connection between state gateways that belong to two different state machines. In contrast to a transition, a signal does not have an ECA rule meaning that it is sent when control-flow enters the output part of a gateway. There are three types of signal that can link two gateways: **spawn** (i.e. creates a new state machine), **finish** (i.e. terminates a state machine) and **message** (i.e. non-terminating). A signal has a lower and upper bound, which are the minimum and maximum number of times it can be sent, i.e. a spawn signal with a lower bound of 1 and upper bound of 5 will create a minimum of 1 object instance and a maximum of 5 object instances. Figure 3.5 is an illustrative example that shows a possible (yet unrealistic) way that the pre-and post-gateways of two states may be connected by four message signals. State 1 has an optimistic pre-gateway and optimistic post-gateway and State 2 has a pessimistic pre-gateway and pessimistic post-gateway. From this it can be derived that several combinations of pre- and post-gateway configurations exist that could cause a deadlock. For example, if State 1 had a pessimistic pre-gateway instead of optimistic, then deadlock will occur because both pre-gateways of State 1 and State 2 will wait in the input part to receive a signal that will never be sent.

The meta-model captures the fact that some types of signals occur in response to, or following another signal. For example, a non-terminating (message) signal *Sig02* can only occur following a spawn signal *Sig01*. Accordingly, the meta-model includes an association that can be created between signals to form a **relationship** between two state machines of different types. A relationship contains all signals that are communicated between two state machines. Specifically, a relationship will link a spawn signal to a finish signal and/or zero or more message signals. The motivation for grouping signals into relationships between state machines is due to the need for a parent state machine to keep track of the number of active children it may have of different types.

The benefits of state machine relationships is not witnessed until runtime. Having access to the relationships that a state machine is involved in allows

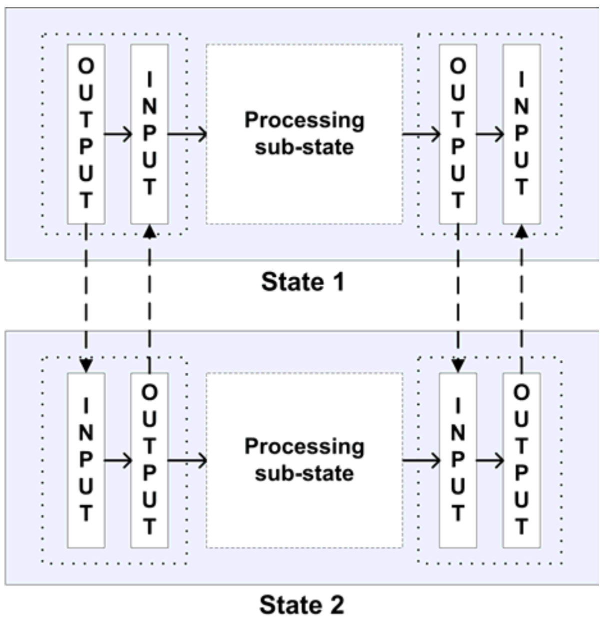


Figure 3.5: Pre- and Post-Gateways Connected by Message Signals

a gateways the ability to block control-flow in a way that is dependent on the gateway mode, as previously discussed. For example, a *wait-for-one* gateway will block until it has received a signal from any source, with the condition that there is at least one active child in any relationship. A *wait-for-rel* gateway will block until all signals have been received from all children from any relationship, with the condition that the gateway is involved in two or more relationships. A *wait-for-all* gateway will wait for all children that have been started in all relationships that the gateway is involved in, with the condition that if no children have been started in any relationship then the gateway does not block control-flow because in this case “no children” is the equivalent of “all children”.

Figure 3.6 presents a concrete graphical syntax for the concepts introduced above. An object is represented by a white rectangle. A state machine is represented by a gray rectangle. The initial and final state of a state machine follow accepted state machine graphical conventions. A state is represented by a rounded rectangle with a pre-gateway and post-gateway located at opposite sides of the state (although gateways can be placed on any available side of a state). Transitions are represented by an arrow with an open arrowhead, where the arrowhead points to the target state. The three types of signal are distinguished by a different arrowhead on a dashed line; a spawn signal is indicated by a double-filled arrowhead, a finish signal is indicated by a double-empty arrowhead and a message signal is indicated by a single filled arrowhead.

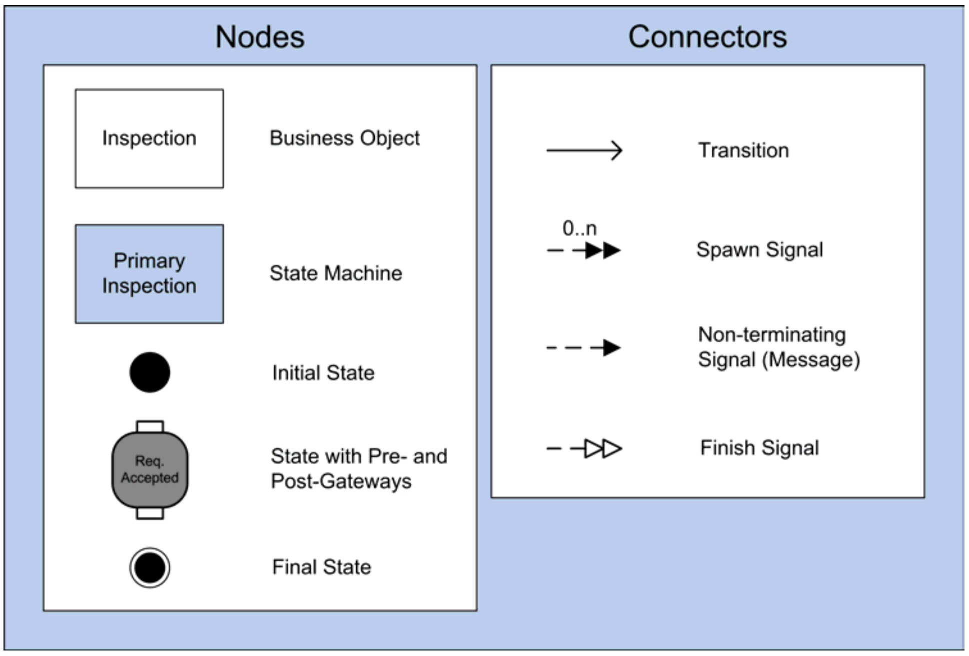


Figure 3.6: O-C Modelling Notation

## 3.2 Working Example – Gas Pipeline Investigation

In this section we will introduce an example that has been modelled using an object-centric approach. The example captures a process snippet of investigation and maintenance for a gas pipeline network provided by a major gas supplier. A gas pipeline network is a geographically extensive system that is subject to regular inspections to uncover issues (e.g. faults, wear and tear, damage, etc.) which must be resolved following a well-defined, quality-controlled procedure, making this an excellent example in which to test an application of object-centric process modelling.

Figure 3.7 shows the cardinality and relations between the classes in the gas pipeline investigation example. The classes in this example are the investigation class, the primary inspection class, the standard issue and critical issue (which are issue subtypes) and the follow-up inspection class (which is a primary inspection subtype).

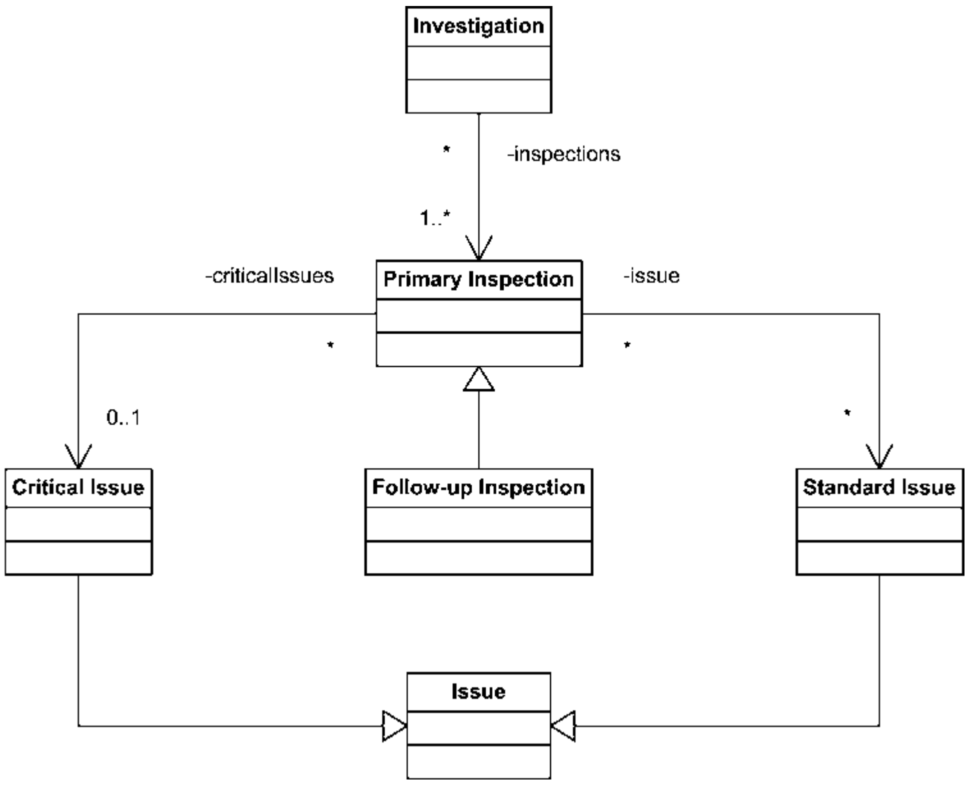


Figure 3.7: Gas Pipeline Inspection Example – Class Diagram

As with most major infrastructure, gas pipelines are routinely investigated following a well-defined procedure. An investigation is conducted between two points on the network (a start point and end point) and there may be two or more types of pipes joined together between the start and end points. The various types of pipes are often of different ages, are constructed of different materials and are operated differently, which means that they are subject to different inspections. Thus an *investigation* consists of one or more *primary inspections* (a primary inspection for each type of pipe) to uncover *issues*. An issue is a problem with plant and equipment that is either malfunctioning or functioning below an acceptable level of performance. Every issue that is discovered during an inspection requires rectification. There are two issue subtypes, a *standard issue* and a *critical issue*. A domain constraint in this example is that a primary inspection can have no more than one critical issue, any other issues found are designated as standard issues. These two types of issues are resolved slightly differently. A *follow-up inspection* may be performed as a quality-control test to re-examine issue rectification, which may in turn uncover additional issues.

Examples of state machine lifecycles that correspond to the primary inspection class, as well as two related state machines corresponding to the critical issue and the issue classes are shown in Figure 3.8. Each class, state machine, gateway and processing sub-state has been given a unique ID, which are shown on the model since we will require them later for the purpose of transforming the model to an activity-centric representation. In this example, a request is accepted to begin an primary inspection. Following acceptance of the request an initial inspection is performed and zero-to-one critical issues and/or zero-to-many standard issues are raised. The primary inspection creates an instance of a standard issue for each issue discovered.

During the resolution of a critical issue, data on the issue is recorded, the issue is resolved and the issue is marked as fixed. As part of the inspection, ratings only need to be collected following the resolution of a critical issue, if one exists. Here we see a need for synchronisation between state machine lifecycles. Ratings are collected after a message signal sent from the post-gateway of the “Resolving Issue” state from the critical issue to the pre-gateway of the “Collect Ratings” state in the primary inspection. However, if no critical issue exists (i.e. the primary inspection did not previously create one), then the primary inspection will not wait for the completion of any critical issues. To achieve this, the pre-gateway of the “Data Loaded” state in the primary inspection is a *wait-for-all* gateway that blocks until a finish

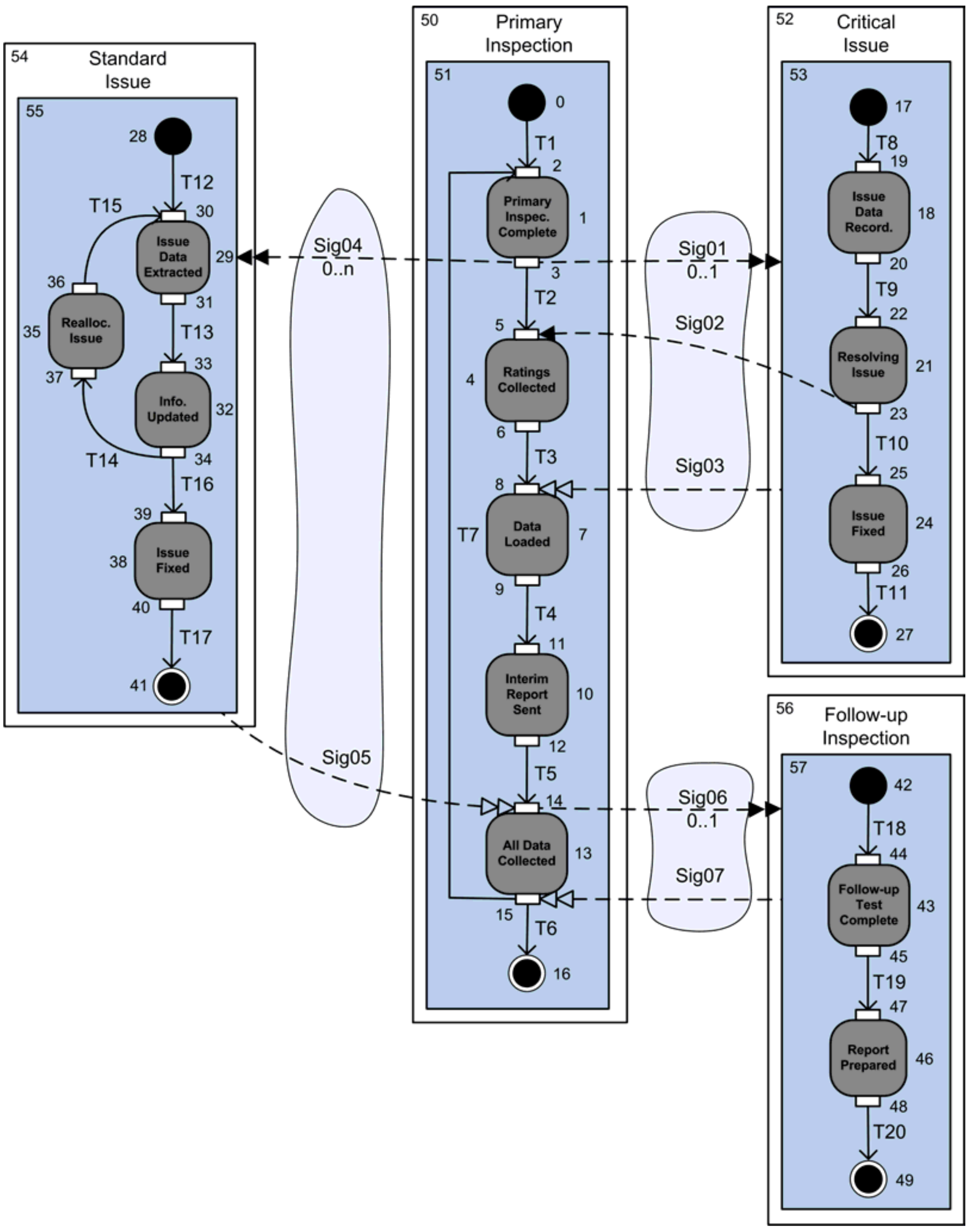


Figure 3.8: Example of an Object Behaviour Model

signal is received from the critical issue. If a critical issue was not created then the gateway does not block, which allows the lifecycle of the primary inspection to proceed.

After an interim report has been sent, data on all issues is collected. This requires that all standard issues have been completed. To achieve this, the configuration and mode of pre-gateway that belongs to the "All Data Collected" state is *pessimistic* and *wait-for-all*, respectively. This means that the gateway will block control-flow until all finish signals sent from standard issues are received before a spawn signal to create a follow-up inspection is sent, if required. The mode of the post-gateway that belongs to the "All Data Collected" state is also *wait-for-all*, meaning that it will block control-flow until a finish signal is received from the follow-up inspection. If no problems are found by the follow-up inspection then the primary inspection is complete. Otherwise, a transition is taken to the Primary Inspection Complete state to deal with the additional issues that have been found.

## 3.3 Transforming Object-centric Models to Activity-centric Models

In this section we introduce a proposal to transform an object-centric model to an activity-centric model. We use the gas pipeline maintenance process as an illustration of this proposal. The essence of the proposal is to analyse an object-centric model in order to extract a set of elementary causal dependencies between events and signals. These elementary causal dependencies are represented as a *causal matrix*, also known as a *heuristics net* [3]. The idea of using a heuristics net comes from the ProM framework [24], where heuristics nets are used as an intermediate representation to construct a Petri net from an event log.

### 3.3.1 Background: Heuristics nets

A heuristics net is composed of a set of transitions, which we call "tasks" to put them in context. Each task is given a unique ID and has an *input* and an *output*, which are sets of task IDs. The input of a task T represents the IDs of the tasks that can start task T. If this set is empty, it means that the task can be started even if no other task has been completed (i.e. this is the initial task in the process model, which is indicated with a '.'). If the input of a task is not empty, it contains one of several *disjunctions*. Each of these disjunctions should be read as an "Or" of several tasks, indicated by a '|'. For example, in Table 3.1, task 34 has a disjunct 37|39 meaning that a choice is made between task 37 or task 39 following completion of task 34.

Symmetrically, the output of a task determines which other tasks can be executed after a given task completes. An empty output (indicated with a '.') denotes a final task. Meanwhile, a non-empty output must be read as a set of disjuncts. For example, a disjunction of the form 37|39 means that either task 37 or task 39 can be executed. An output can contain multiple disjunctions, which are read as an "And" of several tasks, indicated by an '&'. For example, the output of task 3 is 5&55&17, which means that after task 3 completes, task 5 can be executed, and task 55 can be executed, and task 17 can be executed. A sample of the tasks belonging to a heuristics net that has been generated from the object-centric model in Figure 3.8 with some explanatory notes of the task input and output is shown in Table 3.1.

A limitation of heuristics nets is that they can only capture a 'flat' representation of a process, since they cannot capture sub-processes. To overcome this limitation, it is desirable to record the 'humps' (or sub-process regions) in the model. This can be achieved by identifying *sub-process delimiters* in the object model. A sub-process delimiter is defined by the point in an object-centric model where an instance of a state machine lifecycle is created (identified using a spawn signal), the point where a state machine lifecycle is ended (identified using a terminating signal), and also the sub-process multiplicity. The set of sub-process delimiters are then stored and may be recalled to restore sub-processes in an activity-centric model during transformation from an object model, which is introduced in the next section.

The inputs/outputs of tasks can be represented either as expressions composed of & and | operators, or as sets of sets and both representations are equivalent. A heuristics net is basically a set of tasks, with each task being associated to an "input" and an "output". In the rest of this chapter

| Task ID | Input | Output | Notes |
|---|---|---|---|
| 0 | . | 1 | An initial task. |
| 1 | 0\|15 | 3 | Input received from task 0 or 15. |
| 3 | 1 | 5&17&55 | Output sent to tasks 5, 17 and 55. |
| 4 | 5 | 8 | |
| 5 | 3&23 | 4 | Input received from tasks 3 and 23. |
| 7 | 4 | 10 | |
| 8 | 4&27 | 7 | Input received from tasks 4 and 27. |
| 10 | 7 | 13 | |
| 13 | 14 | 15 | |
| 14 | 10&55 | 13&57 | Input received from tasks 10 and 55, output sent to 13 and 57. |
| 15 | 13&57 | 1\|16 | Input received from tasks 13 and 57, output sent to either 1 or 16. |
| 16 | 13 | . | A final task. |
| 17 | 3 | 18 | |
| 18 | 17 | 21 | |
| 21 | 20 | 23 | |
| 23 | 21 | 5&24 | Output sent to tasks 5 and 24. |
| 24 | 21 | 27 | |
| 27 | 24 | 8 | |
| 28 | . | 29 | An initial task. |
| 29 | 28\|35 | 32 | Input received from task 28 or 35. |
| 32 | 29 | 35\|38 | Output sent to task 35 or 38. |
| 35 | 32 | 29 | |
| 38 | 32 | 41 | |
| 41 | 38 | . | A final task. |
| 57 | 14 | 15 | Multiple Instance Composite task delimiter. |

Table 3.1: Heuristics Net Sample from Figure 3.8.

we use set notation to represent the input and output of tasks in a heuristics net, for example instead of 3|(5&6) we write {{3},{5,6}}. Each element in the set is itself a set that represents a given disjunct.

### 3.3.2 Transformation procedure

The transformation procedure consists of the following three steps, which are performed in the order depicted in Figure 3.9:

**I -** Generate a heuristics net from an object model/state machine diagrams.

**II -** Generate a Petri net from a heuristics net.

**III -** Transform the Petri net into a YAWL process model.

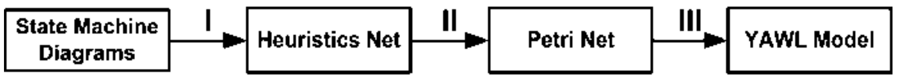


Figure 3.9: Transformation Procedure Overview

Below, we present an algorithm that automates **Step I**. For each state in an object model, this algorithm generates two tasks corresponding to the pre- and post-gateway. In other words, each pre- or post-gateway in the object model will lead to one task in the generated heuristics net.

*Algorithm 1* takes as input an object model and produces the corresponding heuristics net. The algorithm iterates over each state gateway in order to generate an input set (preTask) and output set (postTask). Because there is a one-to-one mapping between state gateways and tasks, the algorithm treats them interchangeably, meaning that the identifiers of gateways in the source object model are identifiers of tasks in the generated heuristics net.

The following auxiliary functions are used in *Algorithm 1*:

- *states* : ObjectModel → Set of State, is the set of states in an object model.
- *pre, post* : State → Gateway, yields the pre or post gateway of a state.
- *inputTransitions, outputTransitions* : State → Set of Transition, yields the set of input/output transitions.
- *source, target* : Transition → State, yields a transition's source/target.

- *inputSignals, outputSignals* : Gateway → Set of Signal, yields a gateway's input/output signals.
- *mode* : Gateway → GatewayMode, yields a gateway's mode.
- *explode* : Set of Signal → Set of Set of Signal. explode({$e_1$, $e_2$, ... , $e_n$}) = {{$e_1$}, {$e_2$}, ... , {$e_n$}}.

**Algorithm 1**: Generation of a heuristics net

```
Input: om : ObjectModel
Output: preTask, postTask : Task → Set of Set of Task
predecessors, successors : Set of Gateway
foreach s ∈ states(om) do
    predecessors := { post(source(t)) | t ∈ inputTransitions(s) };
    successors := { pre(target(t)) | t ∈ outputTransitions(s) };
    preInputSignals := { source(g) | g ∈ inputSignals(pre(s)) };
    preOutputSignals := { source(g) | g ∈ inputSignals(post(s)) };
    postInputSignals := { target(g) | g ∈ outputSignals(pre(s)) };
    postOutputSignals := { target(g) | g ∈ outputSignals(post(s)) };
    if mode(pre(s)) = wait-for-one then
        preTask(pre(s)) := { predecessors, preInputSignals(s) };
    else
        preTask(pre(s)) := { predecessors } ∪
        explode(preInputSignals(s));
    postTask(pre(s)) = { { post(s) }, postInputSignals(s) };
    if mode(post(s)) = wait-for-one then
        preTask(post(s)) := { { pre(s) }, preOutputSignals(s) };
    else
        preTask(post(s)) := { { pre(s) } } ∪
        explode(preOutputSignals(s));
    postTask(post(s)) := { successors } ∪
    explode(postOutputSignals(s));
end
```

To analyse the inbound and outbound causal dependencies of a gateway, each gateway is conceptually decomposed into two parts: the input and the output. The input corresponds to the signals the gateway has to wait for, while the output corresponds to the signals the gateway has to send out.

The input and output sets are generated as follows. The pre-gateway input set is the union of the source of each incoming transition with the source of each incoming signal, depending on the gateway mode (i.e. *wait-for-one* or *wait-for-all*). If the gateway mode is *wait-for-one* then the preTask set is the set of input transition sources (predecessors) and the set of pre-gateway input signal sources. However if the gateway mode is *wait-for-all* then the preTask set is the union of the set of input signal sources converted to a set of set of signals by the *explode* function with the set of predecessors. The postTask set consists of the post-gateway and the targets of all outgoing signals sent from the pre-gateway. The procedure for constructing the input and output sets for a post-gateway is symmetric to the corresponding procedure for a pre-gateway. After the application of *Algorithm 1* a heuristics net is constructed by inserting the input and output set as individual rows in the net.

In **Step II** of the transformation procedure the heuristics net is passed to the heuristics net conversion tool in ProM to obtain a Petri net. **Step III** is performed using a workflow net conversion plugin in the ProM framework to combine the sub-process delimiters with the derived Petri net to create an 'unflattened' YAWL model by including sub-process definitions in the YAWL model as shown in Figure 3.10. This step is merely a syntactic transformation that aims to exploit the process modelling constructs in YAWL because any Petri net can be seen as a YAWL process model, but YAWL allows one to represent certain patterns in a more compact manner.

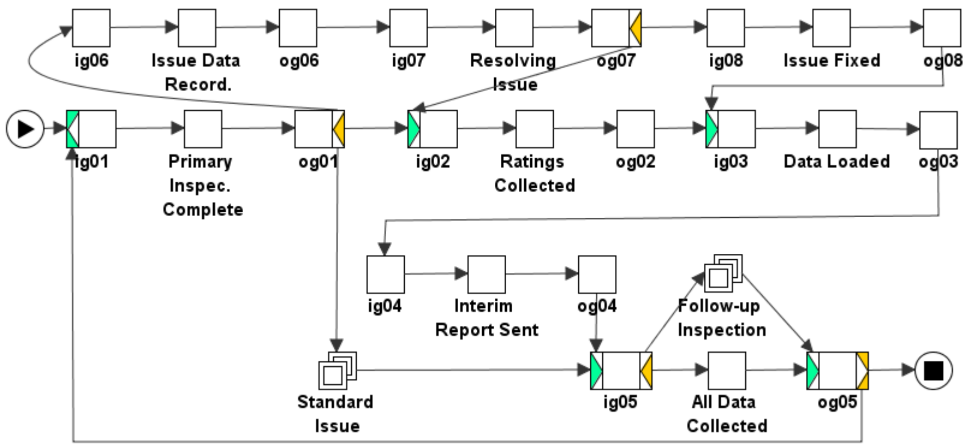


Figure 3.10: Gas Pipeline Inspection Process Model in YAWL

We now discuss the procedure to restore sub-processes in the YAWL model in further details. In the absence of any message signals between parent and child state machine lifecycles in an object model, the region between a spawn signal and the finish signal becomes what is known as a single-entry-single-exit (SESE) region [102]. Since an SESE region and a YAWL sub-process share the characteristic of having a single entry and exit point, an SESE region can be directly mapped to a YAWL sub-process. The start and end points of an SESE region are indicated by sub-process delimiters which are defined as part of the transformation from an object model to a heuristics net. The YAWL lexicon has two kinds of sub-processes; a composite task (i.e. the YAWL construct for capturing sub-processes) and a multiple instance composite task (i.e. the YAWL construct for capturing sub-processes that are executed multiple times concurrently). A composite task can be distinguished from a multiple instance composite task by inspecting the lower- and upper-bound multiplicity recorded by the relevant sub-process delimiter. If the lower- and upper-bound multiplicity equals one then the sub-process is a composite task, but if the lower-bound is zero or the upper-bound is greater than one then the sub-process is a multiple instance composite task.

### 3.3.3 YAWL Model Reduction Rules

After a model has been transformed from an object-centric to an activity-centric representation, a number of tasks exist that have empty decompositions. No work is performed by these tasks, since they are used for control-flow routing only. We hereby refer to these tasks as *control-flow nodes*. These control-flow nodes correspond to gateways in the original model. For example, task "og05" in Figure 3.10 corresponds to the post-gateway of the state "All Data Collected" in Figure 3.8. In *Algorithm 1*, gateways in an object-centric model are converted to tasks for the purpose of preserving control-flow behaviour. These tasks in the heuristics net then become empty transitions when the heuristics net is converted to a Petri net, and then these empty transitions become YAWL tasks with an empty decomposition when the Petri net is converted to a YAWL net. Because control nodes do not perform work it is possible to remove them by means of model reduction rules in order to obtain a simpler yet equivalent YAWL model.

Below, we present four reduction rules that we use to post-process the YAWL nets produced by the above conversion procedure, in order to eliminate tasks with empty decompositions. The following notation is used to represent these reduction rules:

- $\varepsilon$: YAWL task with no decomposition (also called control-flow node).
- $T$: YAWL task that may or may not have a decomposition.
- X: control-flow join decorator.
- Y: control-flow split decorator.

We now define and illustrate the following four reduction rules:

1. Task Input Combination.
2. Task Output Combination.
3. Task Join Combination.
4. Task Split Combination.

In each rule we attempt to remove an *epsilon task* ($\varepsilon$) from the model by combining it with either the previous or subsequent task. Some context conditions exist for the rules to be applied. For the first and second rules, X is a control-flow join decorator that is an AND-join or an XOR-join or an OR-join, Y is a control-flow split that is an AND-split or an XOR-split or an OR-split. For the third and fourth rules, X and Y must be of the same type of split or join decorator, or else the reduction rule cannot be applied.

**1. Task Input Combination**: If a control node $\varepsilon$ is connected via an outgoing arc to a task node $T$, where $T$ has no more than one incoming arc from $\varepsilon$ and $\varepsilon$ has no other outgoing arcs, then the incoming arcs of the join X become incoming arcs to $T$ as illustrated in Figure 3.11. In the case where $\varepsilon$ has only one incoming arc (thereby has no join decorator X) the incoming arc becomes the incoming arc to $T$. For example, in Figure 3.10 "Gatewayig0" has only one incoming arc which will become the incoming arc to the "Req. Accepted" task after applying this rule. It is allowable for the join X to be of a different type to the split Y (e.g. X is an AND-join and Y is an XOR-split). Relocation of the join X to the task $T$ retains the original control-flow behaviour because synchronisation at X always occurs before the tasks $T$ can be executed in both the left and right side of Figure 3.11.

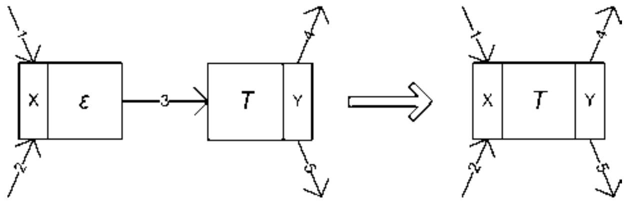


Figure 3.11: Task Input Combination Reduction Rule

**2. Task Output Combination**: If a task node $T$ is connected via an outgoing arc to a control node $\varepsilon$ where $T$ has no more than one outgoing arc to $\varepsilon$ and $\varepsilon$ has no more than one incoming arc from $T$, then the outgoing arcs of the split Y are relocated to become outgoing arcs of $T$, as illustrated in Figure 3.12. As in the Task Input Combination Reduction Rule it is possible that the join X can be a different type to the split Y. Relocation of the split Y from $\varepsilon$ to $T$ retains the original control-flow behaviour because control-flow is split at Y after $T$ is executed in both the left and right side of Figure 3.12.

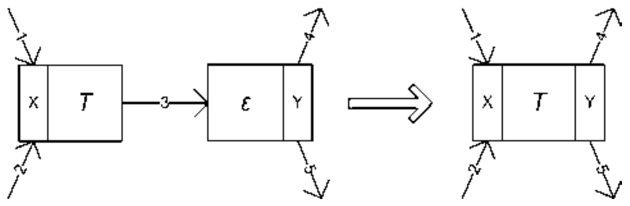


Figure 3.12: Task Output Combination Reduction Rule

**3. Task Join Combination**: If a control node $\varepsilon$ is connected to a task $T$ and $\varepsilon$ has more than one incoming arc and both $\varepsilon$ and $T$ have the same join type (e.g. AND-join), then $\varepsilon$ and $T$ can be combined to become $T$. The incoming arcs to the joins that belong to both $\varepsilon$ and $T$ are combined to become incoming arcs to the join X at $T$ (minus the arc that connects $\varepsilon$ to $T$) as illustrated in Figure 3.13. If $\varepsilon$ has only one incoming arc (i.e. has no join) then this arc is added to the set of incoming arcs to $T$.

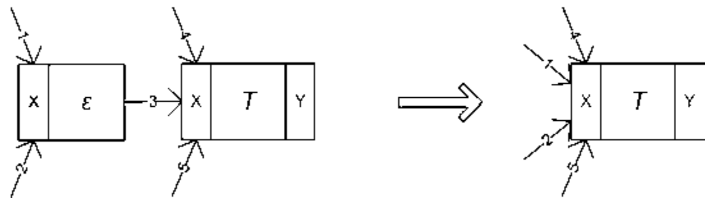


Figure 3.13: Task Join Combination Reduction Rule

**4. Task Split Combination**: If a task $T$ is connected to a control node $\varepsilon$ and $T$ has more than one outgoing arc and both nodes have the same split type (e.g. AND-split) and $\varepsilon$ has no more than one incoming arc from $T$, then the outgoing arcs from $T$ and $\varepsilon$ (minus the arc connecting $T$ to $\varepsilon$) can be combined to become outgoing arcs of $T$ as illustrated in Figure 3.14.

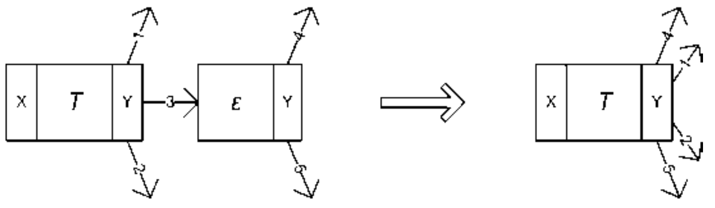


Figure 3.14: Task Split Combination Reduction Rule

Additional considerations are necessary to deal with conditions on the arcs of the resulting split if Y is an XOR-split or an OR-split decorator and also their evaluation order if Y is an XOR-split. The reason for this is that YAWL's treatment of these types of splits is such that they exhibit correct control-flow behaviour no matter what the conditions associated with the outgoing arcs are.

Let us first examine the case where Y is an XOR-split. In YAWL an evaluation order is associated with outgoing arcs of an XOR-split and control is passed at runtime to the first arc of which the condition evaluates to true.

If there is no arc of which the condition evaluates to true then control is passed to the arc that comes last in the evaluation order. Hence care needs to be taken when merging two tasks with XOR-split decorators in order to preserve their behaviour - in particular, if the arc connecting the two nodes to be combined is the last arc in the evaluation order, as this means that at runtime the control-flow could pass this arc if its condition evaluates to true but also if its condition evaluates to false. Referring to this arc as $r$ and its associated condition as $c$, the arcs of the combined XOR-split should be evaluated in the order of the first XOR-split until arriving at arc $r$. The arcs of the second split should then be evaluated in their original order which are then followed by the remaining arcs (if any) of the first XOR-split in their original order.

In case $r$ was not last in the evaluation order, then the conditions of the arcs associated with the second XOR-split should be changed to reflect the fact that at runtime the control-flow could only be passed if condition $c$ evaluated to true as well as their original condition, hence their condition should change to a conjunction of $c$ and their original condition. For example, if the tasks in Figure 3.15 are combined using the Task Join Combination rule where the arcs of the first and second XOR-split are evaluated from top to bottom, then the evaluation order of the combined task is $s$, $u$, $v$, $t$ ($r$ is replaced by $u$, $v$) and the conditions associated with $u$ and $v$ change to become conjunctions consisting of their original condition and the condition associated with r. In case $r$ was last in the evaluation order, then the conditions associated with the arcs of the second XOR-split do not need to be changed because control could pass to the second task in the original model even if $c$ evaluated to false.

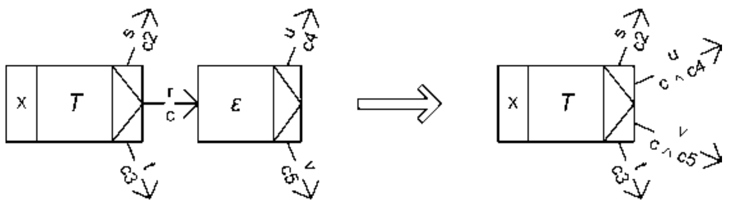


Figure 3.15: Task Split Combination Reduction Rule (XOR-split)

Let us now consider the case where Y is an OR-split. In YAWL an OR-split has a designated outgoing arc which is referred to as the *default* arc to which at runtime the control-flow is passed if none of the conditions associated with its outgoing arcs evaluates to true (including the condition associated with the default arc itself). We will distinguish between the case where the arc $r$ connecting the tasks to be reduced is the default arc and the case where another outgoing arc of the first OR-split is the default arc.

In the former case, the default arc of the second OR-split becomes the default arc of the OR-split resulting from the reduction. Noting that control would only have passed to the second node in the original model if either all conditions associated with the outgoing arcs of the first OR-split evaluated to false or the condition associated with the default arc evaluated to true, the conditions associated with arcs of the OR-split resulting from the reduction need to be modified. Hence conditions associated with arcs that originate from the second OR-split need to change to a conjunction of their original condition and a disjunction consisting of $c$ and a conjunction where the elements are negations of the conditions associated with the outgoing arcs of the first OR-split (excluding $r$). For example, if the tasks in Figure 3.16 are combined using the Task Join Combination rule where $r$ is the default arc of the first OR-split and $u$ is the default arc of the second OR-split, the conditions associated with the arcs $u$ and $v$ become $c4 \wedge (c \vee (\neg c2 \wedge \neg c3))$ and $c5 \wedge (c \vee (\neg c2 \wedge \neg c3))$ respectively, and $u$ becomes the default arc of the combined OR-split.

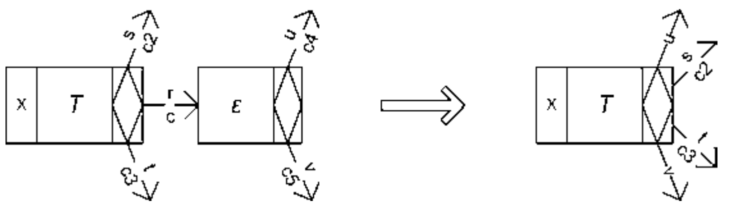


Figure 3.16: Task Split Combination Reduction Rule (OR-split)

The situation is considerably simpler in case $r$ is not the default arc of the first OR-split. In that case the OR-split resulting from the reduction inherits the default arc from the first OR-split. The conditions of the arcs originating from the second OR-split need to reflect the fact that control would only have been passed to them in the original model in case not only their original condition evaluated to true but condition $c$ as well. Hence the condition of these arcs are a conjunction of their original condition and $c$.

Certain combinations of control nodes and tasks cannot be reduced. Two examples are; a control node $\varepsilon$ that has a split which is connected to a task $T$ that has a join; and a task $T$ that has a split which is connected to a control node $\varepsilon$ that has a join, shown in Figure 3.17. Reduction rules are not applied to these cases because an attempt to reduce such task and control node combinations will alter control-flow behaviour.

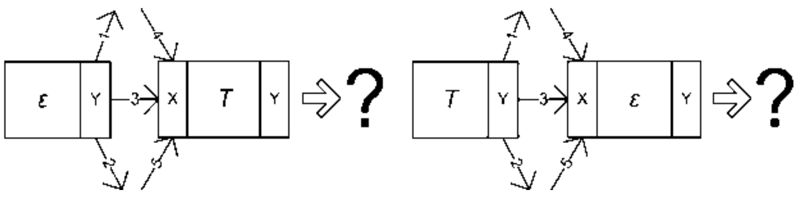


Figure 3.17: Example Combinations Where Reduction is not Applied

Since the target modelling notation in our case is YAWL there are several elements in the YAWL lexicon that we make some assumptions about. Firstly, there are no cancellation regions in the activity-centric model because a cancellation in a state machine is performed by taking a transition to a state named 'Cancelled' or similar, therefore no explicit "cancellation region" exists in the generated YAWL model. Secondly, implicit conditions are the only kind of condition found in the activity-centric model except for the start and end conditions, which are preserved by the reduction rules to ensure that a single entry and single exit point exists for every process model (including sub-processes).

The application of the model reduction rules to the YAWL model from the gas pipeline inspection example (Figure 3.10) is shown in Figure 3.18. In this example it has been possible to apply the reduction rules to merge almost all of the control nodes (pre- and post-gateways) with tasks. The control-flow behaviour has been maintained and the model is clearer to read.

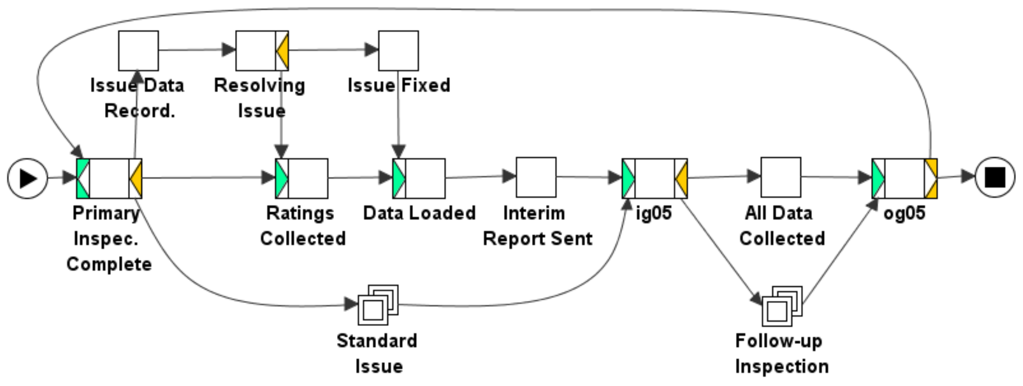


Figure 3.18: Inspection Process Model in YAWL Following Reduction Rules

It is worth noting that there exist other model reduction rule-sets in the literature: these include Murata's Petri net transformation rules [61] and Wynn's state-space reduction rules [115]. Murata's rules allow one to transform a Petri net while preserving soundness: if the original net is sound, the net resulting after applying a transformation rule is also sound. Murata's rules can also be used to eliminate tau-transitions (i.e. transitions without labels) in a net while preserving its semantics. However, Murata's rules are defined on Petri nets and need some adaptation to be applicable to YAWL nets. This is the idea followed by Wynn *et al* [115] who propose a set of reduction rules for cutting down large YAWL nets to ease verification. Again, these rules could potentially be used to eliminate tasks with empty decompositions in YAWL models. However Wynn's reduction rules are considerably more complex than what we need to post-process YAWL nets produced by our transformation. This is the reason why we designed a specific set of reduction rules to this end.

## 3.4 Tool Support

The proposal including the model reduction rules has been implemented as a tool programmed in Java on top of the Eclipse platform.[1] The tool, known

[1]`http://www.eclipse.org/platform/`

as *FlexConnect*, is a graphical editor for creating object models. It also contains a module that enables the automated transformation of state-based object behaviour models to YAWL nets, based on the algorithm presented in Algorithm 1. The implementation of the tool relies on libraries from the ProM framework[2] to perform the transformation from a heuristic net to a Petri net and from a Petri net to a "flat" YAWL net (i.e. the model contains no multiple instance tasks). Subsequently, these YAWL nets are "unflattened" by an extension to these libraries. The YAWL nets are then reduced according to the reduction rules as explained above.

The modelling tool, the model transformation technique and the reduction rules have been tested using the gas pipeline inspection and maintenance example, which is a real-life process model provided by the industry partner. A screenshot of the gas pipeline inspection and maintenance example which has been modelled in FlexConnect is shown in Figure 3.19. The process captured by the model in this scenario is a larger version of the model presented in Figure 3.19 and provided us with an example of a standardised, well-defined process to test the process modelling concepts that were introduced in this chapter. An extension to these concepts to cater for flexible process is introduced in Chapter 4.

To test the modelling tool, a subprocess of the whole pipeline inspection and maintenance model was chosen that consisted of 76 states and 15 objects. This subprocess specifies and coordinates the work involved in inspecting and repairing the inner concrete lining of a particular type of gas pipeline. The entire example (the parent of this subprocess) coordinates the work involved in inspecting and repairing seven different types of gas pipelines, thus it is approximately seven times larger than this. However, given the size and complexity of the chosen subprocess, it was decided that this subprocess would provide a sufficient test scenario for the modelling tool and transformation algorithm.

[2]`http://www.processmining.org`

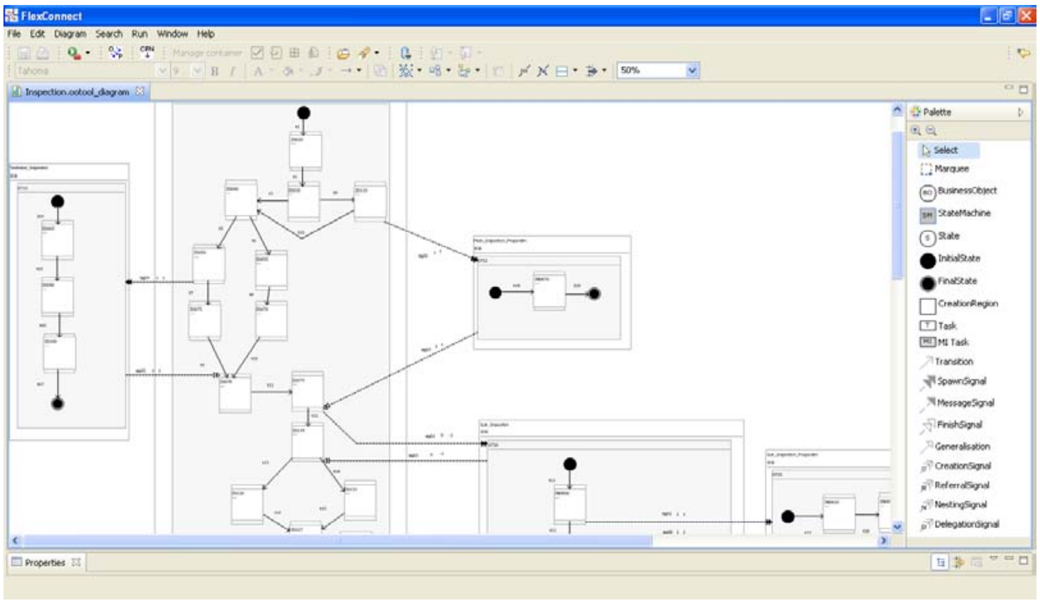


Figure 3.19: Screenshot of the Inspection Process Model in FlexConnect

## 3.5 Summary and Discussion

Modelling Information Systems using object-oriented techniques is a mainstream design approach. Mainstream object-oriented analysis and design practices (e.g. those based on UML) are based on concepts of objects whose structure is captured as classes and whose behaviour and interactions are captured as state machines, sequence diagrams and similar modelling notations. In this chapter a meta-model for the purpose of designing O-C process models was introduced and then discussed in detail. The purpose of defining the meta-model is to identify the salient aspects of O-C process modelling and create a foundation upon which O-C process models can be developed. An O-C modelling notation was introduced and its use was illustrated using an industrial example concerned with inspection of gas pipelines.

Having established an O-C meta-model and modelling notation, the second half of the chapter was concerned with detailing a transformation procedure and an algorithm to bridge the differences between O-C and activity-centric process models in terms of control-flow logic. Specifically, we define a transformation between the proposed O-C meta-model and YAWL. The conclusion of this exercise is that it is possible to transform O-C process models (defined in the proposed notation) into YAWL, so long as the objects only interact through spawn and finish signals. When intermediate message signals are involved the transformation can still be done, but the resulting YAWL model will be "flat" meaning that the decomposition captured in the O-C process model is lost in the transformation. A question that remains following model transformation is a proof of semantics preservation. Such a proof would require one to produce the set of traces or the state space for each modelling construct, and to show that this set of traces/state space for the original construct in FlexConnect and the translated YAWL construct are identical. Providing this proof is out-of-scope of this thesis.

To validate the transformation procedure the algorithm was implemented into an O-C modelling tool called *FlexConnect* which was comprehensively tested using the gas pipeline investigation and maintenance example to generate an equivalent model in the YAWL notation. Our proposal is a step towards bridging O-C models and process models that is intended to be applied during the design phase of the project development lifecycle.

We observe that a reverse transformation, i.e. from an activity-centric to an object-centric representation of a process is also possible, but has proven to be rather difficult to accomplish in practice. The problem of performing this transformation is that an activity-centric model must contain a significant amount of object-centric meta-data to identify the object types, object boundaries and object input/output(s). Previous work on this problem has been investigated by Kumaran *et al* [50], who proposes an algorithm to discover the business entities ("information-centric" artifacts) in an activity-centric model. However, precise control-flow dependencies *between* objects are missing in the resulting object model (i.e. Which object creates what other object(s)? Which tasks synchronise between objects? What type of synchronisation is needed?). The outcome is that the process model structure is lost after the transformation from activities to objects. Work by Küster *et al* [52] also investigated linking the lifecycles of activity-centric models and state-based object models for the purpose of evaluating object model consistency. Although the structure of either model is retained, problems with this approach are that two types of models must be established and maintained, plus the definition of hard links between these models makes the approach inherently inflexible. To advance the transformation proposal in this chapter to a bi-directional model transformation, an activity-centric meta-model and/or modelling notation is required that allows an object to be more than a data storage mechanism.

In the next chapter we will investigate extensions to the base meta-model for the purpose of enabling flexibility in O-C process models.

# Chapter 4

# Object-centric Process Model Extensions for Flexibility

Mainstream business process modelling techniques often promote a design paradigm wherein the activities that may be performed within a case, together with their usual execution order, form the backbone on top of which other aspects are anchored. This Fordist paradigm, while effective in standardised and production-oriented domains, breaks when confronted with processes in which case-by-case variations and exceptions are the norm. We contend that the effective design of flexible processes calls for a substantially different modelling paradigm. Motivated by requirements from the human services domain, we explore the hypothesis that a framework consisting of a small set of coordination concepts, combined with established object-oriented modelling principles provides a suitable foundation for designing highly flexible processes. Several human service delivery processes have been designed using this framework and the resulting models have been used to realise a system to support these processes in a pilot environment [75]. The framework is presented in this chapter and we show how it is used to address different flexibility requirements using a series of illustrations.

Process-Aware Information Systems, such as traditional Workflow Management Systems, have difficulties supporting dynamic business processes because they rely on modelling paradigms that tend to impose a given execution order between activities and decision points. This fact has been discussed in the literature for some time leading to many proposals for *flexible workflow* support [107, 70, 105, 21]. In this chapter we demonstrate how to capture highly flexible business processes using an object-centric process modelling approach. The approach is inspired by, but arguably not limited to, the delivery of human and social services. Modelling and executing processes in this domain presents additional challenges compared to other more standardised domains such as insurance and banking. A key feature of delivering human and social services is that the type, number and order of tasks and sub-processes needed to address a case are often not known until runtime. Also, variations on a case-by-case basis and exceptions are the norm in these processes. An attempt to impose a standard way of delivering social services is usually met with resistance by the stakeholders involved in the process – both from the providers and consumers of social services.

In this chapter, we explore the hypothesis that an object-centric modelling approach provides a suitable basis for capturing the extreme levels of process flexibility needed to manage human social services. The main contribution is a meta-model for the design of highly flexible processes based on object-oriented concepts. The meta-model has been embodied in a modelling tool that allows us to design O-C process models. This chapter is based on work published in [75].

## 4.1 Patterns of Flexibility

In our experience in applying object-oriented approaches to design process-aware systems that need to deal with ad-hoc situations, a range of requirements have been observed that are condensed into three *patterns of flexibility*. A pattern of flexibility is a recurrent problem wherein a designer needs to account for the fact that a variety of circumstances could be encountered during the execution of a process model, yet the scope of these circumstances needs to be captured at design-time to achieve some uniformity (since an organisation provides a finite number of services) or to enforce certain constraints. Each pattern of flexibility also involves a class of users (e.g. social workers or case managers). For convenience these patterns of flexibility are referred to as PoF1-PoF3.

### 4.1.1 PoF1: Creation Flexibility

Creation flexibility is the ability of a user to trigger the creation of one or more task instances (*jobs*) in an unplanned manner during execution of a process. This pattern of flexibility allows the set of task types to be instantiated as well as the ordering of instantiations to be loosely specified at design-time. Creation flexibility is similar to the case handling approach [8] where tasks do not need to be performed in a strict order and do not necessarily have to be completed to complete a case (meaning that the tasks are optional). At the same time, it is necessary to define constraints regarding the number of task instances and/or the state(s) in a process where unplanned task instances can be created.

Generally speaking, a task instance is created in either a *planned* or an *unplanned* manner. A *planned* task is created *as-specified* by process model logic. An *unplanned* task presents additional concerns since it is created *on-demand*, i.e. if and when the task is required. For example, a *Health Assessment* task may require additional tasks that correspond to subtypes of *Treatment*, but the additional treatments are difficult to completely plan at design-time because the treatment(s) depend on the assessment.

### 4.1.2 PoF2: Delegation Flexibility

Delegation flexibility is the ability of a user to trigger the transfer of context and data from an executing task to a different task. This pattern of flexibility provides support for circumstances that may change over time (i.e. if a problem appears during a client interaction, delegate the interaction to a task that can support the problem). Due to circumstances that frequently affect the delivery of human social services, situations regularly occur that require the context and data from a task to be fully transferred to another (specialist) task.

To support such situations, a new task (*delegatee*) takes over execution of a previous task (*delegator*). For the purposes of control-flow, a delegatee replaces a delegator, meaning that when a delegatee completes, the completion is treated as if the delegator had completed. This feature, together with the fact that data is fully transferred from the delegator to the delegatee, distinguishes delegation flexibility from creation flexibility. The delegation relation is transitive, meaning that a delegatee may also transfer its execution to another task.This feature, along with the fact that data is transferred from a delegator to a delegatee, distinguishes delegation flexibility from creation flexibility. Note that from a data-flow perspective, the delegatee is a subtype of the delegator, since the delegatee needs to receive as input the data collected by the delegator and to produce as output at least the same data as the delegator.

### 4.1.3 PoF3: Nesting Flexibility

Nesting flexibility is the ability of a user to create instances of nested sub-processes as they are needed. For example, during execution of a homelessness process a social worker may discover an additional major issue with the client concerning an alcoholism issue which is well beyond the scope of the original process that manages homelessness issues. Similar to (task) creation flexibility, nesting flexibility is sometimes only allowed under certain constraints (e.g. the number of sub-processes can be bounded or unbounded and the type of sub-processes can only be created in designated states of a process). However, nesting flexibility deals with creating sub-processes rather than creating tasks – we call this situation a *referral*. This pattern of flexibility enables a system to create as many layers of ad-hoc sub-processes as needed to manage issues as they arise, while maintaining sub-process modularity and retaining process control.

## 4.2 Elements for Flexible Object-centric Processes

We aim to fulfill the objective of achieving flexibility in object-centric models by proposing a design framework consisting of three abstract types of business objects, namely the Coordination Object, Job Object and Referral Object. These objects are used to construct process models that can capture the patterns of flexibility (PoF1-PoF3) introduced in the previous section. In this section we describe the properties and interactions of these objects.

We propose to achieve process flexibility via an extended meta-model that consists of three abstract types of business objects, namely Coordination Object, Job Object and Referral Object. As shown in Figure 4.1, a concrete business object type inherits from an abstract type.

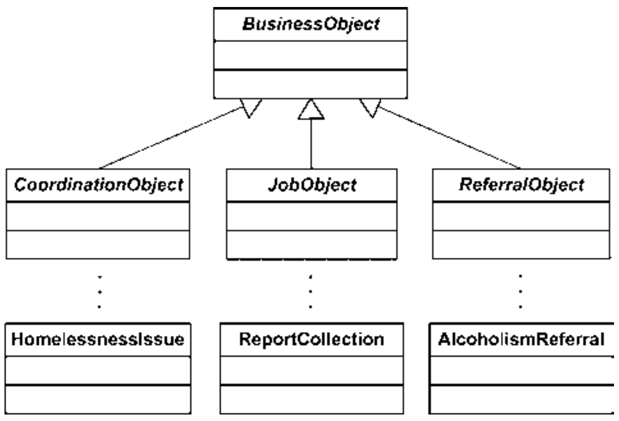


Figure 4.1: Abstract Types and Concrete Types

A **Coordination Object** (COROB) is an object that *coordinates* a process. The COROB is inspired by the recognition that a clear separation must be made between the tasks managed by a process and how the tasks are connected. The net outcome is known as coordination, which explains how this object gets its name. A COROB is responsible for both the creation and synchronisation of the tasks needed to complete a process, managing the execution of a process as well as referring out of scope work to other COROBs.

A **Job Object** (JOB) is an object that represents a task. A JOB manages task execution and reports task completion to its parent object. For example, two JOBs in the social services process model are the *Report Collection* and *Client Visit* which both have the *Client Intake COROB* as their parent.

A **Referral Object** (ROB) is an object that allows a COROB to refer a situation which is outside of its scope to another COROB. For example, if several unplanned major issues appear during the execution of a *Homelessness COROB* such as an *Alcoholism* or *Drug Dependency* issue, a ROB is created that operates under the guidance of a user to create a COROB.

The base meta-model of object types and their relations was shown in Chapter 3 (specifically, Figure 3.1) as an ORM. The flexibility extensions to the base meta-model are also captured using ORM and is presented in Figure 4.2. We now introduce the base meta-model extensions.

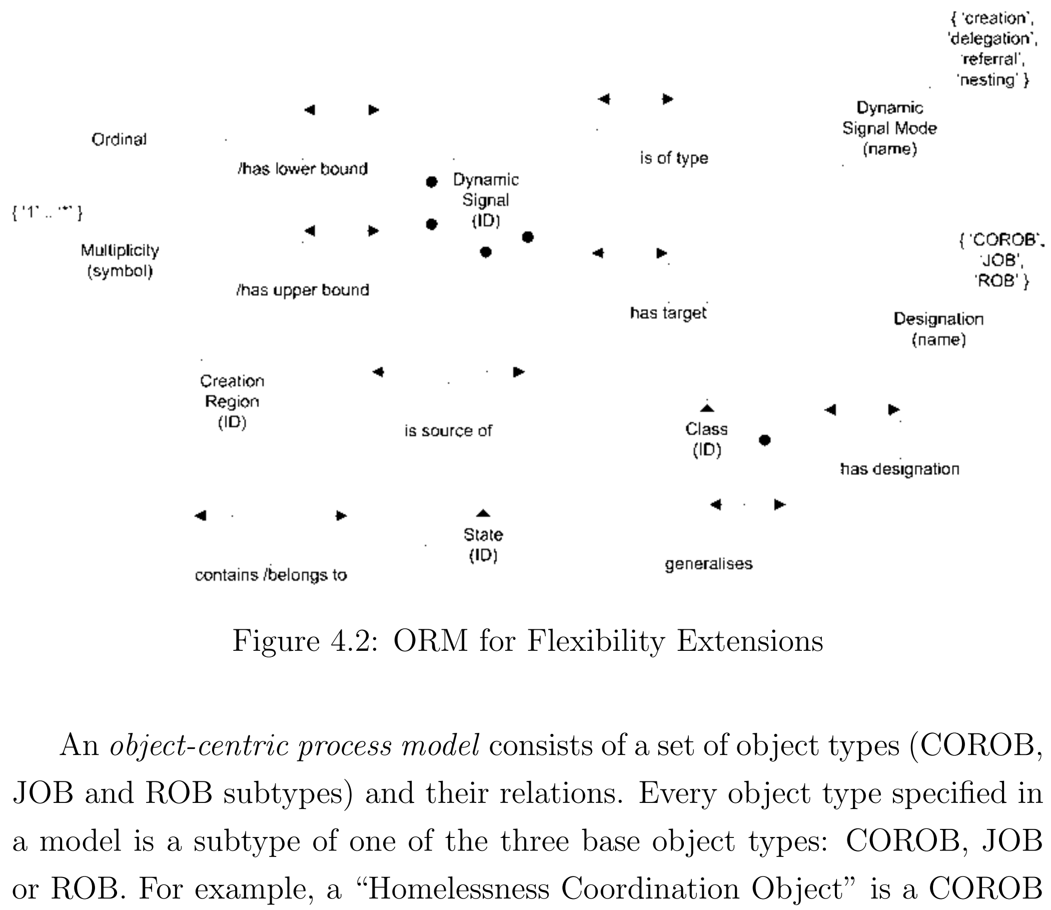


Figure 4.2: ORM for Flexibility Extensions

An *object-centric process model* consists of a set of object types (COROB, JOB and ROB subtypes) and their relations. Every object type specified in a model is a subtype of one of the three base object types: COROB, JOB or ROB. For example, a “Homelessness Coordination Object” is a COROB subtype and a "Client Appointment" is a JOB subtype. A subtype relation is established by using a generalisation association. Generalisation is a classical object-oriented concept that allows a subtype to inherit attributes and behaviour from a supertype. In the case of object-centric process models we may make use of generalisation to define an object hierarchy that share common attributes and behaviour. For example, a "Skin Treatment", "Eye Treatment" and "Mental Health Assessment" JOBs are subtypes of a "Treatment" JOB. The generalisation association allows a supertype to delegate its lifecycle to a subtype at runtime and requires that each subtype sends and receives the same signals as a supertype and sends and receives at least the same data as its supertype, while allowing the subtype to capture an object lifecycle that specialises the supertype. Since the "correct" application of behavioural specialisation of object lifecycles (i.e. ensuring that inheritance does not lead to behavioural inconsistencies) is a separate research question and has been covered in works (for example) by Schrefl and Stumpter found in [91] and [92], we do not elaborate any further on this topic.

A *creation region* is a collection of one or more states in a state machine from within which it is possible to create object instances from a set of object types. A state can belong to more than one creation region, but those states must belong to the same state machine. From a creation region, any number of dynamic signals can be sent. A *dynamic signal* allows a process designer to model object communications that *may* occur (in contrast to *static signals* that model communications which *will* occur, as introduced in Chapter 3), meaning that users have the possibility of triggering a dynamic signal, but they may or may not do so. The source of a dynamic signal is a creation region and the target is an object type. If the state of a source object is within the creation region, users are offered the possibility to trigger the dynamic signal. When the dynamic signal is triggered, an instance of the target object type (or one of its subtypes) is created. The target object type depends on a selection strategy associated to the dynamic signal and input given by the user when triggering the dynamic signal. This approach follows the principle of the Strategy Pattern [28].

There are four dynamic signal subtypes: the delegation, creation, referral and nesting signal. A *delegation signal* allows delegation from a creation region within a source delegator JOB to a target delegatee JOB. A delegator may delegate to more than one type of delegatee, which must be a subtype of the delegator. A *creation signal* enables instances of a JOB to be created from a creation region. The difference between delegation and creation signals is the following. When a delegation signal is triggered, the source object ceases to exist and is replaced by the target object. Meanwhile, in the case of a creation signal, a new target object is created and the source object continues to exist. A parent-child relationship is then established between the source object and the newly created object.

Creation and delegation signals serve to transfer control to a JOB. On the other hand, referral and nesting signals serve to transfer control to a COROB. A user may trigger a *referral signal* if an issue arises during the execution of a COROB that falls outside the scope of the COROB. The newly created ROB then assists users in finding a suitable COROB type to address the issue in question. During the execution of a ROB, a user (not necessarily the same who created the ROB) may then trigger a *nesting signal*, resulting in the creation of a new COROB to handle the issue in question. In Figure 4.3, we show how the COROB, JOB and ROB can be connected using the four dynamic signal types to capture our three PoFs.

## 4.3 Working Example – Social Service Provision

As a motivating scenario, we consider a process executed in the context of a charity organisation. A recently homeless family contacts a charity and makes an application for assistance. The charity opens a case to manage the family's homelessness issue. During the management of the homelessness case it is discovered that there are additional alcoholism and gambling issues that individual family members require assistance with. Each of these issues

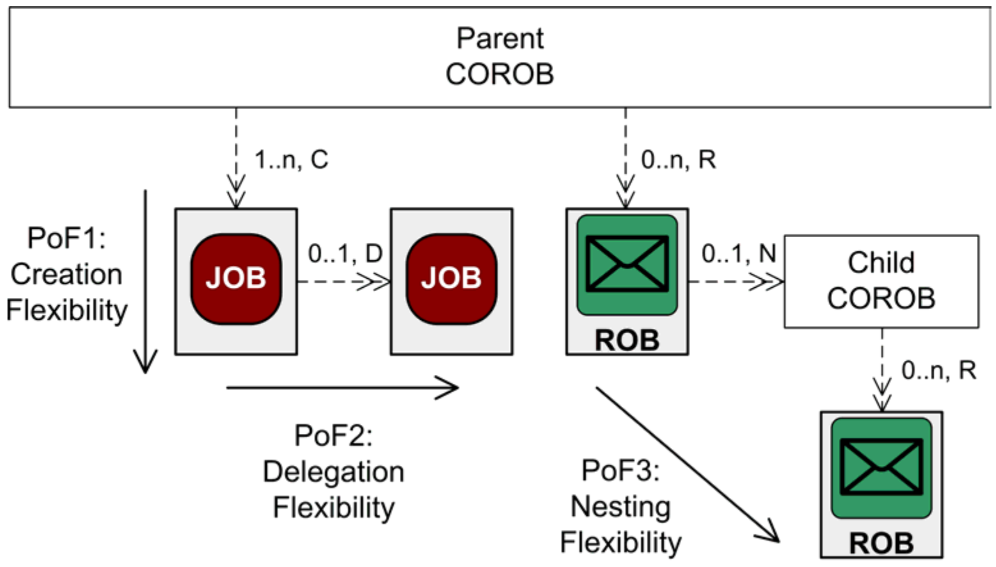

Parent COROB
1..n, C
0..n, R
PoF1: Creation Flexibility
JOB
0..1, D
JOB
ROB
0..1, N
Child COROB
0..n, R
ROB
PoF2: Delegation Flexibility
PoF3: Nesting Flexibility

Figure 4.3: Patterns of Flexibility in the Framework

can be mapped to a social service that are offered by the charity, but the actual delivery of these services remains *unplanned*. An unplanned situation is particularly challenging to capture using traditional process modelling notations due to the possibility that several potential execution scenarios for a single process model must be captured at design-time. A system that can coordinate unplanned situations requires a framework which supports several types of flexibility but can also enforce constraints where necessary. The elements of the framework are represented graphically using the notation in Figure 4.4.

In this section we demonstrate how the framework elements can be used to design a flexible process. For purposes of illustration we refer to a social service process for a charity organisation that has been modelled using the object-centric approach presented in this chapter, which is presented in Figure 4.5. This model consists of a Client Intake COROB that manages the process of accepting new clients who have contacted the charity for assistance. The COROB is responsible for creating and coordinating the tasks and sub-processes involved in new client intake such as completing a risk assessment, visiting the client and collecting reports from social workers, whilst

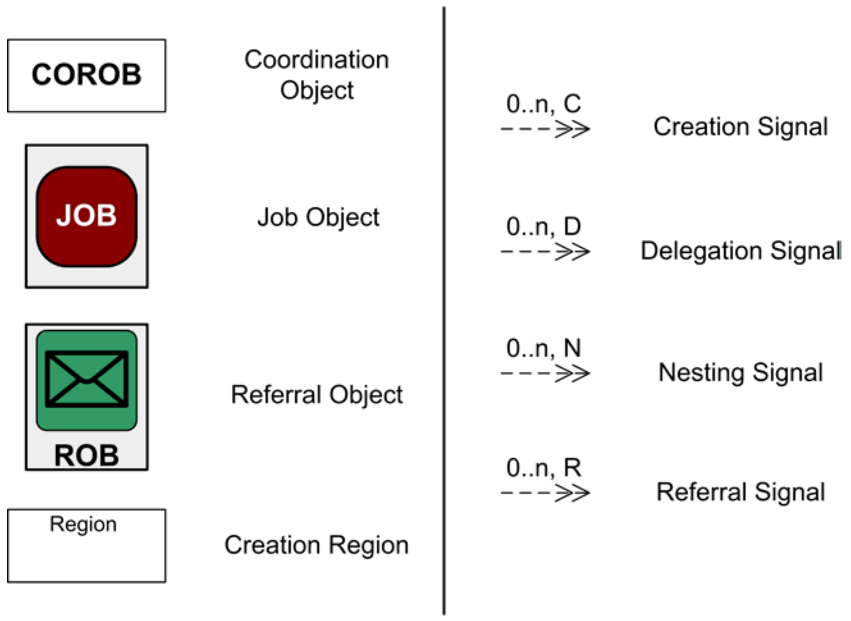


Figure 4.4: Extended Object Model Elements

also coordinating distribution of major issues to other COROBs. The model captures several points in the process where flexibility is either allowed or constrained. For example, a referral to a Homelessness COROB can be performed at any time in the Review Region but at no other time. To counter the possibility of a variety of exceptional circumstances arising at runtime the model has been designed to capture the creation, delegation and nesting patterns of flexibility. The rest of the section uses extracts of the process model shown in Figure 4.5 in order to discuss how the framework addresses the three patterns of flexibility.

### 4.3.1 Demonstrating Creation Flexibility

Creation flexibility is achieved by specifying the set of JOBs that can be created on-demand by defining a creation region within a COROB then linking the creation region to those JOBs with the *creation signal*, as shown in Figure 4.6. In this example a social worker tailors a plan for a client to resolve the issue(s) that the client is faced with. Since the plan is tailored to

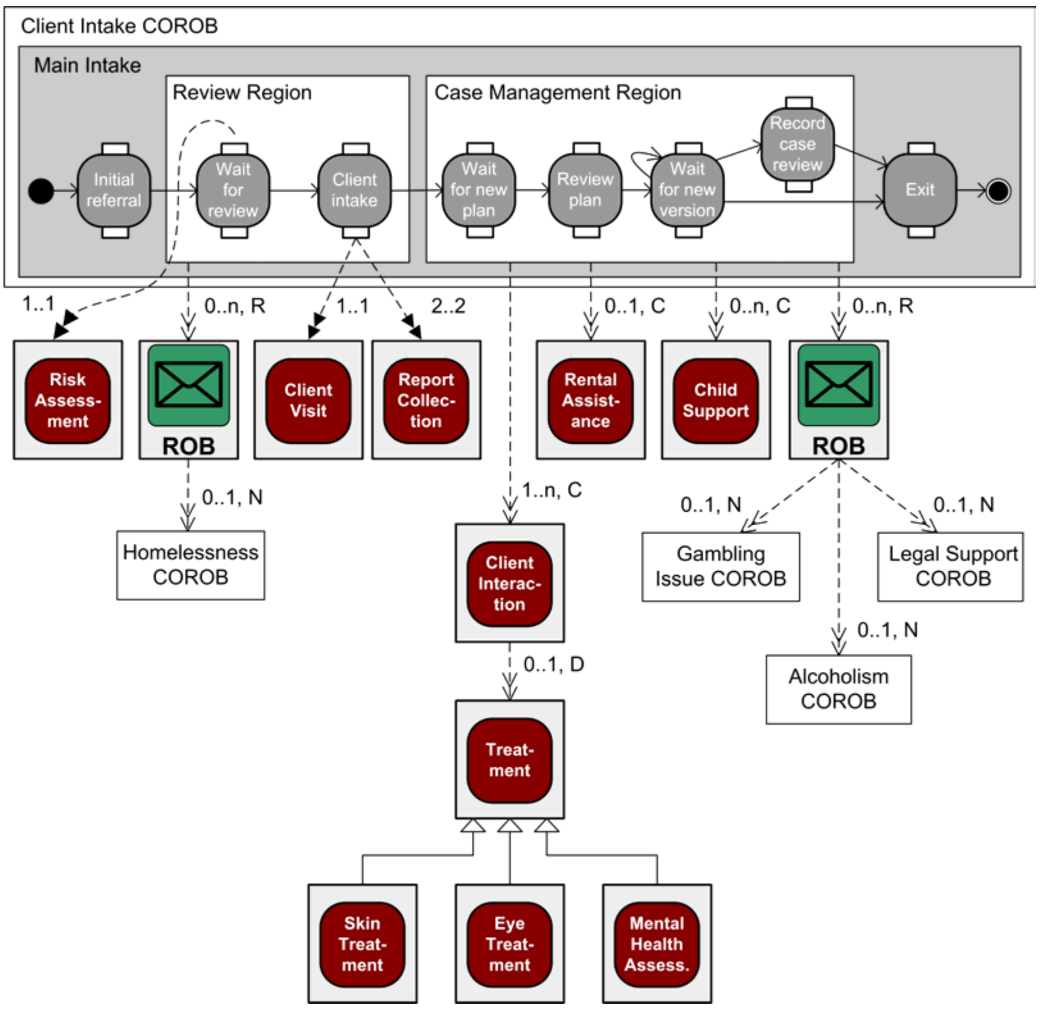


Figure 4.5: Object-centric Social Services Delivery Model

the unique circumstances of an individual, the plan for each client is almost always different. To operationalise the plan the social worker then requires access to different tasks offered by the charity (represented by the JOBs). Creation flexibility gives the social worker the ability to create instances of a task when it is needed (i.e. in any of these states: “Wait for new plan”, “Review plan”, “Wait for new version” and “Record case review”), rather than when it is planned.

When the Client Intake COROB is in a state contained in the Case Management Region, *1..n* instances of the Client Interaction JOB, *0..n* instances

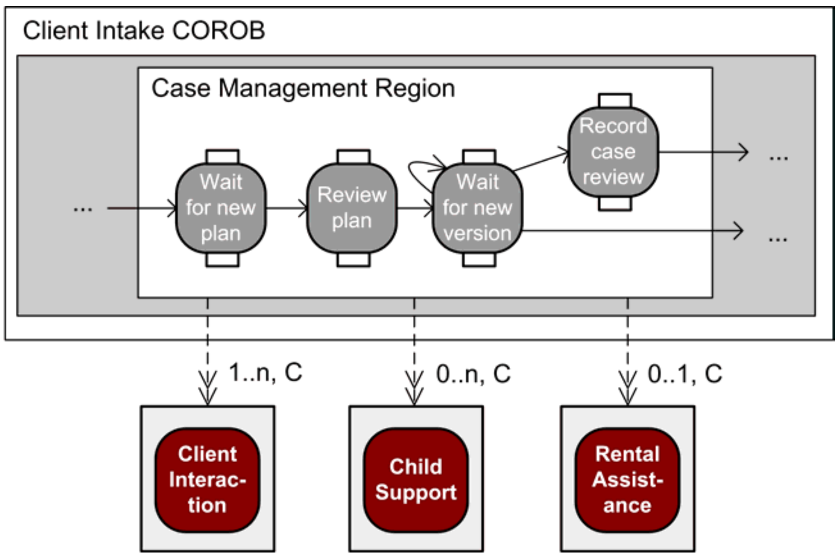


Figure 4.6: Creation Pattern of Flexibility

of the Child Support JOB and *0..1* instances of the Rental Assistance JOB can be created. At least one Client Interaction JOB *will* be created before exiting the Case Management Region, but more than one instance *may* be created. Any number of Child Support JOBs along with a maximum of one Rental Assistance JOB *may* be created. Creation flexibility allows a designer to capture on-demand task creation while also constraining the type and number of task instances according to the business rules.

### 4.3.2 Demonstrating Delegation Flexibility

Delegation flexibility is achieved by linking a creation region in a JOB to one or more tasks using the *delegation signal*. In Figure 4.7, we demonstrate delegation using the Client Interaction delegator JOB. This JOB contains three states ("Make appointment", "See client" and "Assessment") and one creation region (named "Assessment Region") that contains the "Assessment" state. This creation region imposes two restrictions on the Client Interaction JOB. Firstly, delegation from a Client Interaction can only be performed when it is in the Assessment Region. Secondly, the set of allowable delegatee tasks from this creation region are the *Skin Treatment*, *Eye Treatment* and

*Mental Health Assessment* JOBs which are subtypes of the Treatment JOB. Delegation is an optional action – a user will make the choice at runtime

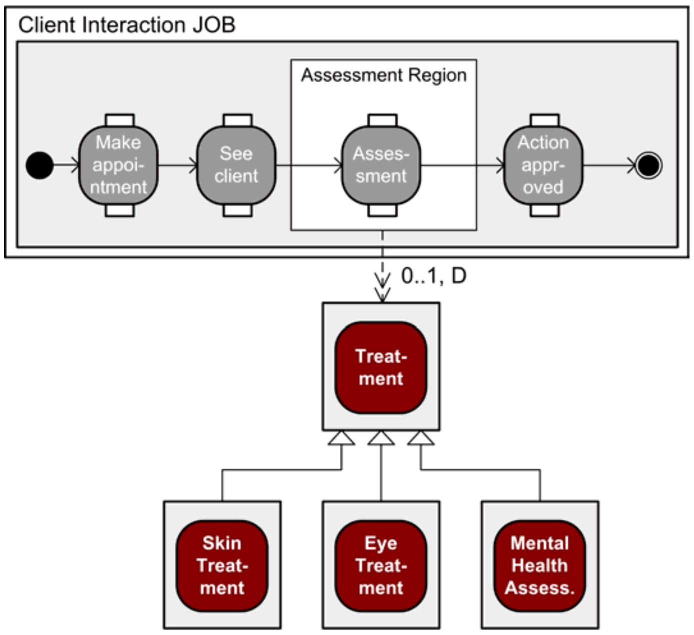


Figure 4.7: Delegation Pattern of Flexibility

of whether or not delegation is performed because the multiplicity of each delegation signal is *0..1*. If a delegator has more than one delegatee then a choice is made by the user to select which JOB will become the delegatee. Delegation can never be mandatory, i.e. a delegation signal must have a lower bound of 0. Delegation is not allowed if the upper bound is greater than 1 because this implies creating clones of the delegator. If multiple instances of a delegator are needed they would firstly be created and then permitted to delegate as required. In case delegation does not occur during the execution of a delegator then its execution will complete normally.

This example illustrates how object inheritance is used to capture delegation associations between tasks in a process model. However we point out that delegation extends the concept of inheritance since at runtime a delegatee must take the data and context of the delegator and must also complete its lifecycle in the same way that the delegator would have.

### 4.3.3 Demonstrating Nesting Flexibility

Nesting flexibility is achieved by linking a creation region in a COROB to a ROB using the *referral signal*, then linking a creation region in the ROB to one or more COROBs using the *nesting signal*. At runtime, a parent COROB may invoke the referral signal to create an instance of a ROB. The ROB may invoke a nesting signal to create an instance of a child COROB to manage the newly discovered real-world issue. The type of child COROB to create is determined by a user. The ROB creates two levels of indirection between the parent and child COROB, giving the framework two advantages.

Firstly, COROBs are decoupled, which establishes COROB modularity. Secondly, the ROB provides the opportunity for human intervention in a referral, since referring major issues between in this manner often needs an approval from a third party resource (e.g. a manager), who can either permit or deny creation of a new COROB instance. Hence, the ROB behaves as an arbiter that separates a parent COROB from its children, allowing children to execute in parallel and allowing a third party resource to maintain control over nested COROBs.

In Figure 4.8 we see the number of referral signals that may be sent from the Case Management Region to a ROB is unbounded (*0..n*) and the ROB is connected to three COROB types. For example, if a social worker discovers an alcoholism issue with a client, a ROB will be created in the system which will in turn create an Alcoholism COROB instance. Alternatively, if an alcoholism and gambling issue are discovered with a client the system will create two ROBs and (given management approval) one ROB will create an Alcoholism COROB and the other will create a Gambling Issue COROB.

The framework places no restrictions on the levels of nesting meaning that a child COROB can in turn create its own ROBs, which can create their own COROBs and so on. For example, as shown in Figure 4.9, in the “Wait for new plan” state an issue resolution plan is prepared for an unemployed client which identifies an unemployment issue beyond the scope of the Client Intake COROB. The issue is referred to a nested Work Search

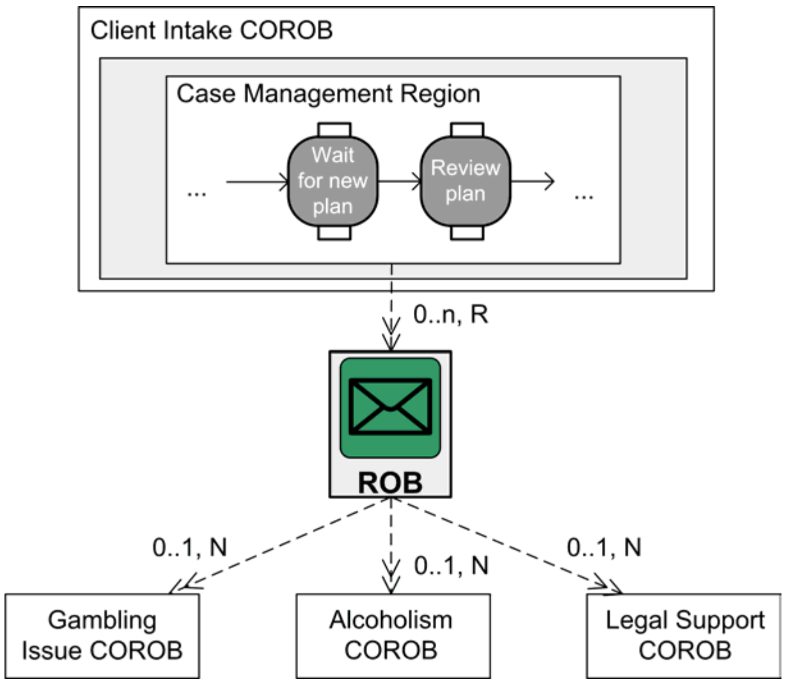


Figure 4.8: Nesting Pattern of Flexibility

COROB. However, during execution of the Work Search COROB the client unexpectedly falls into serious trouble with the police. The Work Search COROB creates a new ROB, which creates a nested Legal Support COROB to support the clients unemployment issue.

We observe that the main benefit of nesting flexibility for a user is the ability to call in different sets of resources and skills in response to situations as they arise. Nesting flexibility allows a COROB to maintain control over the type and number of all dependent COROBs without being directly linked to them, while also establishing an unplanned structure of nested processes. Using the examples in this section we have demonstrated how an O-C process model can handle unplanned tasks and issues. The modelling notation is based on an object behaviour meta-model that has been designed to approach exceptional circumstances as they occur by engaging creation, delegation and nesting flexibility. The ability to handle work in the different ways that it may appear is the point of distinction which allows several flexibility requirements that were identified in Section 4.1 to be supported.

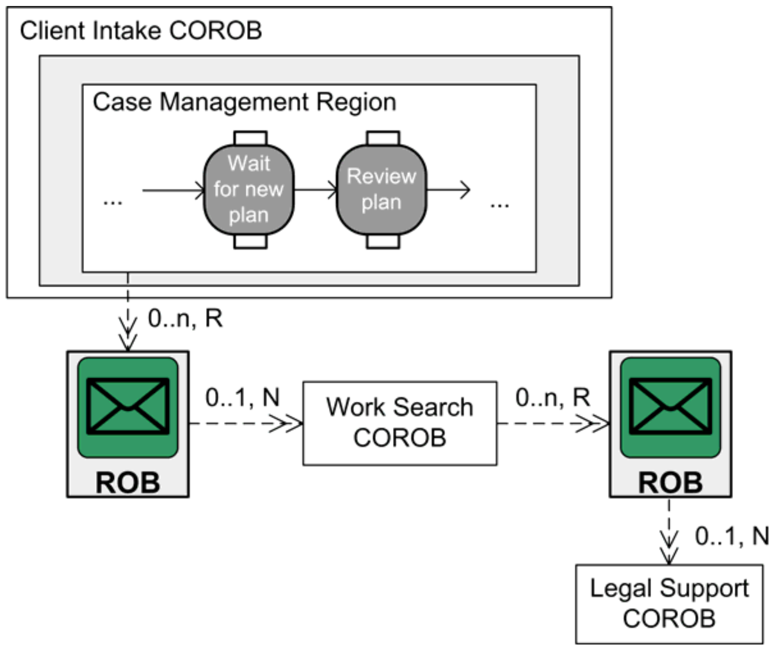


Figure 4.9: Nested Unplanned Sub-processes

The concept of creation regions in particular enables a designer to clearly define which types of flexibility are related to which set(s) of states. This approach gives a process model designer the ability to express that flexibility *is* required at particular points and that flexibility *is not* required at other points, which is beneficial for the design of flexible process models. In the next section we present a tool called *FlexConnect* that supports modelling of flexible object-centric models as presented in this chapter.

## 4.4 Tool Support

A modelling tool named *FlexConnect*[1] has been developed that allows us to design O-C process models as described in this chapter. FlexConnect is a tool that assists process designers to develop O-C process models. FlexConnect was developed using the Eclipse Graphical Modelling Framework (GMF). The foundation of the tool is the UML Class diagram shown in Figure 4.10

[1]FlexConnect can be downloaded from `http://code.google.com/p/flexconnect/`

that specifies the FlexConnect GMF Domain Model. The GMF Domain Model is a specification of the modelling elements and their associations.

The modelling tool has an extension that generates and exports an initial marking to a file that is used as input to a CPN. The CPN that was developed for this purpose is discussed in Chapter 5. which provides the ability to formally check, validate and simulate the behaviour of models that have been designed using FlexConnect. The modelling tool, export function and the generated CPNs have been tested with 20 sample O-C process models of varying sizes in order to evaluate the behaviour of the elements of the base model as well as validate each pattern of flexibility. This includes the social services example presented in this chapter (see Figure 4.5) which is shown as a FlexConnect model in Figure 4.11.

We will now walk through this social services support model at runtime. Upon entering the Review Region the“Wait for review” state is entered. A Risk Assessment JOB is completed for the applicant while an initial application is being completed. At this stage it is either confirmed or not that the client has a Homelessness Issue. A Homelessness Issue is a major issue that requires management by a separate COROB that was designed to manage such an issue. If a Homelessness Issue is confirmed, the Main COROB refers this new work out to a ROB which creates a nested instance of a Homelessness COROB.

Following creation, the Homelessness COROB will execute in parallel to the Main COROB, creating its own tasks that manage the needs of the client to do with their homelessness issue. During the execution of the Homelessness COROB an additional issue is discovered with the client to do with a drug dependency. The Homelessness COROB reacts to this issue by invoking a referral. The RO is guided by the user to create a nested instance of a Drug Dependency COROB that executes in parallel to the Homelessness COROB. This parallelism is handled in a structured manner due to the concept of nesting flexibility.

After the “Client intake” state is entered, three tasks are created. A Client Visit JOB is created along with two Report Collection JOBs. The

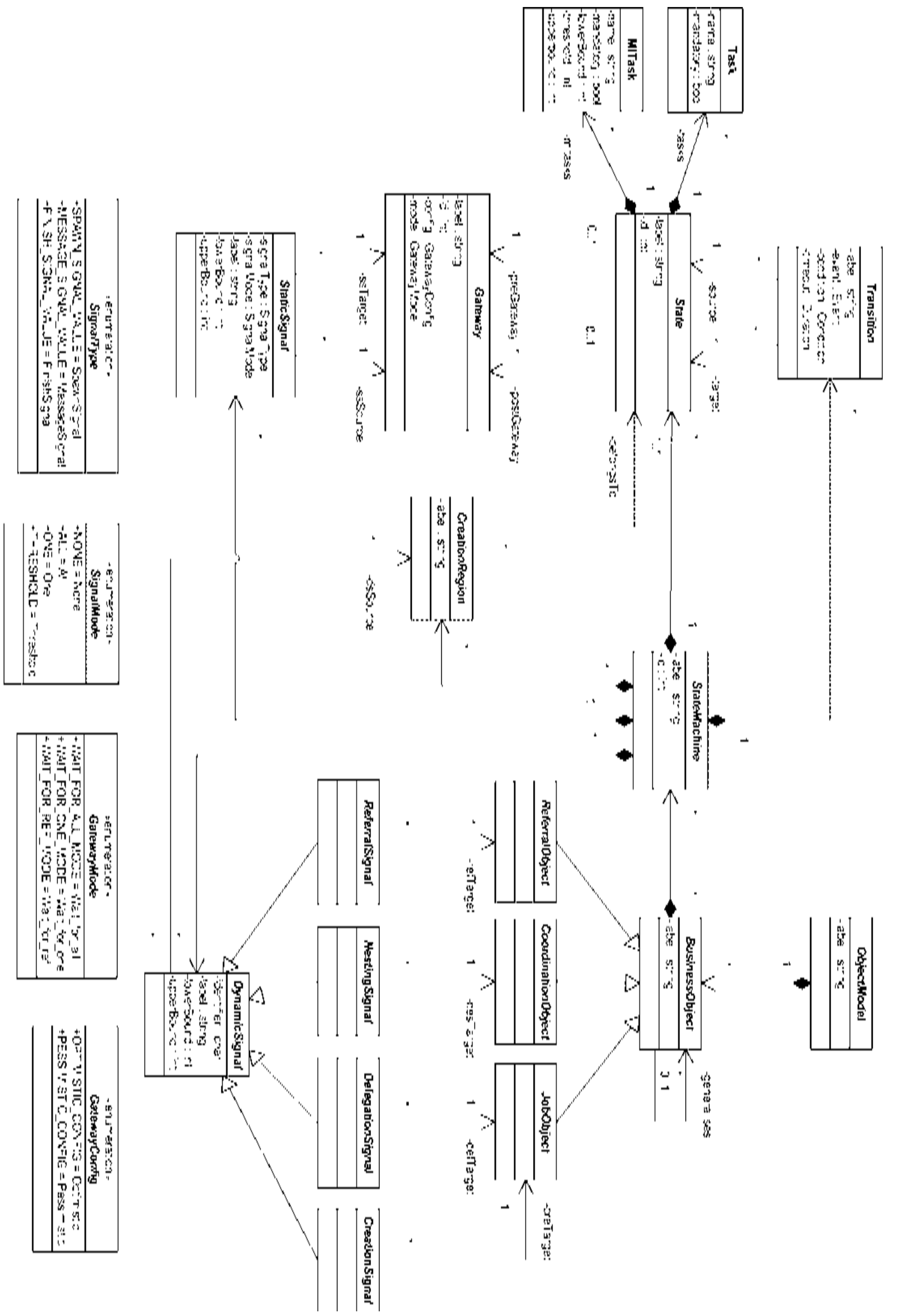


Figure 4.10: UML Class Diagram for Object-centric Flexibility

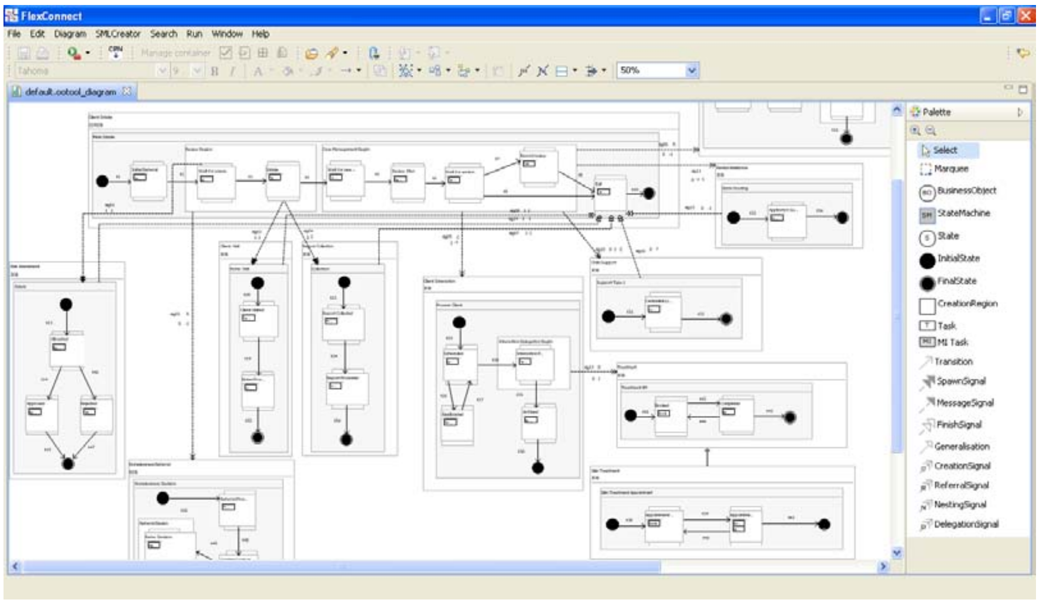


Figure 4.11: Social Services Model in FlexConnect

Client Visit manages the procedure of a social worker's visitation to a client, while the Report Collection manages the work involved with reporting on the recovery progress of a client.

After exiting the "Client intake" state the Review Region is exited and the Case Management Region is entered. This region consists of four states, which are: "Wait for new plan", "Review plan", "Wait for new version" and "Record case review". In any state of the Case Management Region we have the ability to create *1..n*, on-demand, Client Interaction tasks. Specifically, at least one Client Interaction TO *will* be created before the Case Management Region is exited, but more *may* be created. This is an example of creation flexibility. In the "Wait for new plan" state a social worker prepares a goal-action plan for the client, which is revised in the "Review plan" state, and a Client Interaction TO is created by the social worker to suit the social workers need to approach the client with clarifications regarding the case. During the interaction with the client the social worker finds that the client needs additional medical care and the Client Interaction is delegated to a Skin Treatment. Here we see an example of delegation flexibility.

During the “Wait for new version” state a major alcoholism-related issue is discovered. To handle this situation an instance of an Alcoholism Issue COROB is created. The creation of this new COROB is performed using the same method as the Homelessness Issue COROB, as this method allows us to manage the uncertainty surrounding the unknown and unpredictable runtime aspects of the process. These unknown aspects are the elements of a process that *may* be invoked, such as an alcoholism issue in this case. The motivation behind supporting the invocation of process elements in this manner is due to the unknown aspects of *if* and *when* during the execution of a Homelessness Issue (and indeed, any other process that supports a social service) that may be encountered.

During the “Record case review” state another major issue is discovered with the client and an instance of a Gambling Issue COROB is created to handle the issue. The ability to handle work in the different ways that it may appear is the point of distinction that allows the flexibility requirements that were identified in Section 4.1 to be supported.

The output of a valid model constructed using the FlexConnect modelling tool is a Standard ML (SML) [99] file. An SML file generation function is found on the FlexConnect toolbar that creates an SML file from an O-C model by pressing a button named “SML Creator”. Upon pressing this button, the syntax of the object model is validated. To avoid creating an invalid SML file the O-C model must pass a series of validation checks. If one or more of the checks are not passed, a list of the problems that were found in the model are presented in a popup box and an SML file is not created. Otherwise, the result is reported in a popup box and an SML file is created. The checks that are performed on a model include:

- The names of all nodes except Tasks (State Machines, States, Gateways and Creation Regions) must be unique and non-null.

- The names of all connections (Transitions, Static Signals and Dynamic Signals) must be unique and non-null.

- The upper bound of all (static and dynamic) signals must be greater than or equal to the lower bound.
- The upper bound of all multiple instance tasks must be greater than or equal to the lower bound.
- Each gateway must have a configuration and a mode.
- Each message signal and finish signal must have either a parent spawn signal or parent dynamic signal.

The SML file contains an initial marking for the following places in the CPN: Signal Connections, Gateway Mode, Gateway Configuration, State Gateways, Transitions, Creation Regions, Dynamic Connections, Generalisation Associations and Tasks. Each place is populated by making a call to a function in the SML file. E.g. the Transitions place calls the *getTransitions()* function, which places a single token in the Transitions place that contains a list of the transitions in the O-C process model. Successfully loading the SML file into the CPN without receiving any error reports indicates that the O-C process model is at least syntactically correct, because the type of each place in the CPN is directly mapped to a concept in the O-C meta-model. For example, the "Signal Connections" place contains a list of static signals in the O-C process model and the "Creation Regions" place contains a list of the creation regions.

## 4.5 Summary and Discussion

In this chapter we demonstrated how a small set of coordination concepts, in combination with established object-oriented modelling techniques, enables the design of highly flexible processes consisting largely of *unplanned* activities. In particular we demonstrated how a small set of object types (i.e. Coordination Object, Job Object and Referral Object) can be combined to capture different patterns of flexibility. The key principle is that a Coordination Object defines "what can happen during a case", rather than "how should it happen". Any constraints regarding which objects can or should be created and when, are overlaid on top of the basic object model. This is in contrast with mainstream process modelling paradigms based on flowchart-like notations, in which the activities to be performed and their control-flow relations form the backbone of a process model.

As previously discussed, the main aspect of flexibility of interest is design-time models that allow the ability to capture the creation of new objects with the intent of performing unplanned activities at run-time. Of course, while flexibility is essential in domains such as human services, there are situations where this flexibility should be constrained. The proposed framework supports the definition of thresholds to constrain the minimal and maximal number of JOB and ROB objects of various types that should be started under a COROB of a given type (cf. the multiplicity constraints of a signal).

In addition to this feature, one may need to define more sophisticated constraints. For example, situations have been encountered that necessitate the definition of *creation regions*. A creation region allows a model designer to establish when instances of a given JOB or ROB type can be created under a COROB of a given type – e.g. a ROB corresponding to "Work Search" COROB should only be started after the "Health Treatment" tasks have completed. Also, situations can occur where one needs to constrain the number of JOBs or ROBs of different types that need to complete before a COROB object moves to a completion state – e.g., a COROB to handle a case for a homeless family will not complete until the process created to deal with their homelessness situation has closed.

The *FlexConnect* modelling tool enables process designers to create O-C process models. A formalism of the approach was presented as a CPN. To provide object models to the CPN, an export function was added to FlexConnect that creates an SML file which can be loaded into the CPN. Details of the CPN are discussed in the next chapter.

# Chapter 5

# Formalisation

The object-centric meta-model and the flexibility extensions presented in the previous chapter have been formalised using a Coloured Petri Net (CPN) [40]. The CPN notation is a graphical notation with a formal semantics that consists of places and transitions, which are connected with a directed arc. A place can hold tokens of a particular primitive type or complex type, (known as a colour set) which are specified using CPN-ML, the functional language for CPN. Transitions perform the transfer of tokens between places, which serves to alter the state of the CPN. CPN is a well-known as a suitable notation to capture concurrency. Capturing the concurrent execution of object lifecycles and synchronisation of the parent-child relationships between objects is a major concern in an object model, which makes CPN an ideal method to define the control-flow semantics of O-C process models.

We have used the CPN notation to formally capture the behavioural semantics of the FlexConnect language. This enables us to define the behavioural aspects of both structured and flexible O-C process models. Modelling the behavioural semantics using CPN provides a formal grounding to the proposal and enables us to formally describe the functionality that drives an O-C process model. The CPN in this chapter was designed and implemented using the CPN Tools software, which allows us to take advantage of the execution facilities of CPN Tools to test the runtime behaviour of object-centric process models. We begin by introducing the Base Model concepts (from Chapter 3) in CPN and then the Extended Model concepts (from Chapter 4).

## 5.1 Base Model in CPN

A state-transition machine diagram that shows the states and transitions of an object lifecycle is shown in Figure 5.1. The passage of thread tokens through this state-transition diagram are the main subject of discussion in this section. At a particular moment in time, an object instance can be in one of the following five core states: (1) Gateway Entered, (2) Optimistic Gateway, (3) Pessimistic Gateway, (4) Gateway Exited or (5) Processing Sub-state.

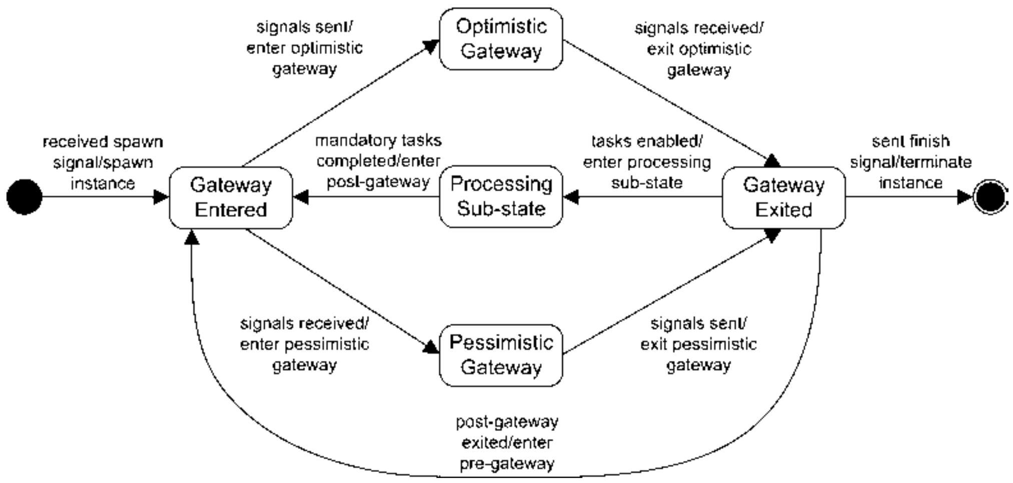

signals sent/ enter optimistic gateway
Optimistic Gateway
signals received/ exit optimistic gateway
received spawn signal/spawn instance
mandatory tasks completed/enter post-gateway
tasks enabled/ enter processing sub-state
sent finish signal/terminate instance
Gateway Entered
Processing Sub-state
Gateway Exited
signals received/ enter pessimistic gateway
Pessimistic Gateway
signals sent/ exit pessimistic gateway
post-gateway exited/enter pre-gateway

Figure 5.1: State-Transition diagram of an Object Lifecycle

To show how object models are executed a series of CPN excerpts are presented. The states shown in the State-Transition diagram in Figure 5.1 form the core of an object lifecycle. Each of these states is directly mapped to a place in the CPN because these states represent the status of an object instance. A token in the CPN that represents an object instance is known as a *thread token*, since the execution of an object is managed by a thread at runtime. A thread token $t$ is a 3-tuple $(n, i, j)$ where $n$ is the name of the gateway or state that the instance currently exists in, $i$ is the instance ID and $j$ is the parent instance ID.

### 5.1.1 Root Object

A *root object* is an object that is not created by another object in an object model, meaning that it has no parent. An instance of an object model is created when an instance of the root object is created. This is accomplished in CPN by firing the "Create New Root Instance" transition as shown in Figure 5.2. An object ID $c$ is obtained from the "Current ID" place which is incremented to ensure that the ID is unique for the next instance to be created because all objects are identified using their ID. The unique object instance ID rule holds for all objects except for instances of a *root object*. The parent object instance ID of a root object is given the value of 0. This instance ID is a reserved root object identifier. The name "is01" is also a reserved name since it is reserved for the initial state of the root object in an object model. When the new token is placed in the "Enter Gateway" place, a new thread token with the value *("is01",c,0)* that represents the root object of an object model is created.

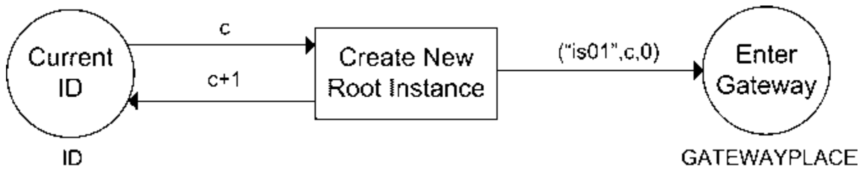


Figure 5.2: Creation of a Root Instance

### 5.1.2 Optimistic and Pessimistic Gateways

The thread token of a newly created instance is initially placed in the "Enter Gateway" place. At this point the thread token will either move to the "Optimistic Gateway" or "Pessimistic Gateway" place, depending on the *gateway configuration*. Each gateway in an object model has a token that specifies the gateway configuration in the “Gateway Config” place, which is a 4-tuple *(g,c,i,f)* where *g* is the gateway name, *c* is the gateway configuration (either *optimistic* or *pessimistic*), *i* indicates if the gateway is the pre-gateway of the initial state and *f* indicates if the gateway is the post-gateway of the final state.

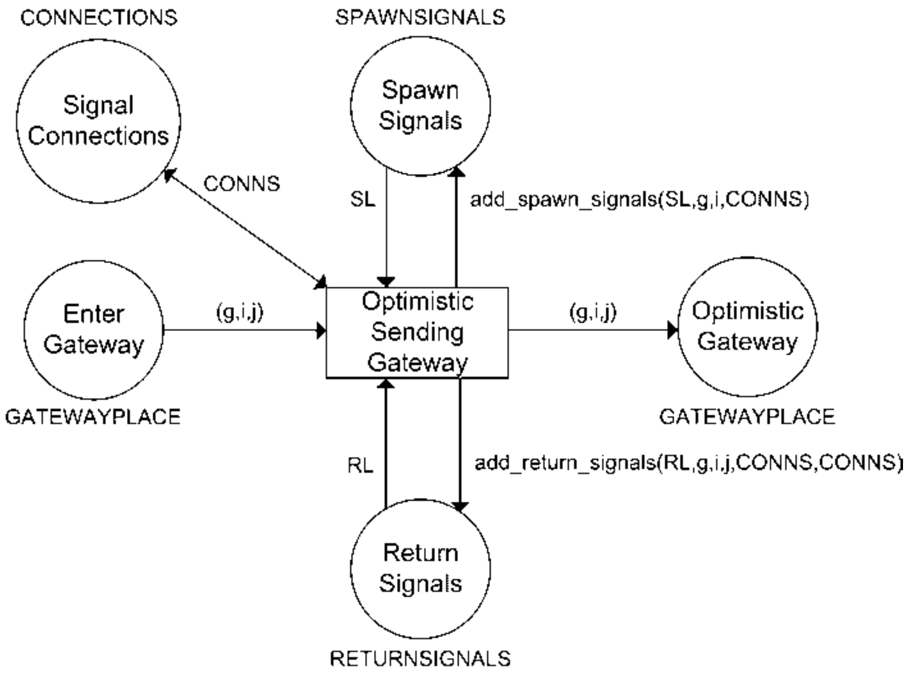


Figure 5.3: CPN - Sending Signals from an Optimistic Gateway

There are two places that contain a thread token when the object life-cycle is currently in a gateway. These are the “Optimistic Gateway” and “Pessimistic Gateway” places. The difference between these two places is the order with which signals are sent and received by the gateway. An optimistic gateway sends signals first, as shown in Figure 5.3. When a thread token is in the “Enter Gateway” place and the gateway is optimistic, the “Optimistic Sending Gateway” place is enabled. When fired, this transition sends all signals from the gateway and moves the thread token to the Optimistic Gateway place.

If the gateway configuration is *optimistic* this means that the gateway sends all signals before waiting to receive signals. In this case the “Optimistic Sending Gateway" transition in Figure 5.4 will be enabled. When this transition is fired, the object instance token is moved from the "Enter Gateway" place to the "Optimistic Gateway" place and two signal sending functions are performed. The gateway can send both *spawn signals* using the *add_spawn_signals* function and non-spawn signals which are known as *return signals* using the *add_return_signals* function.

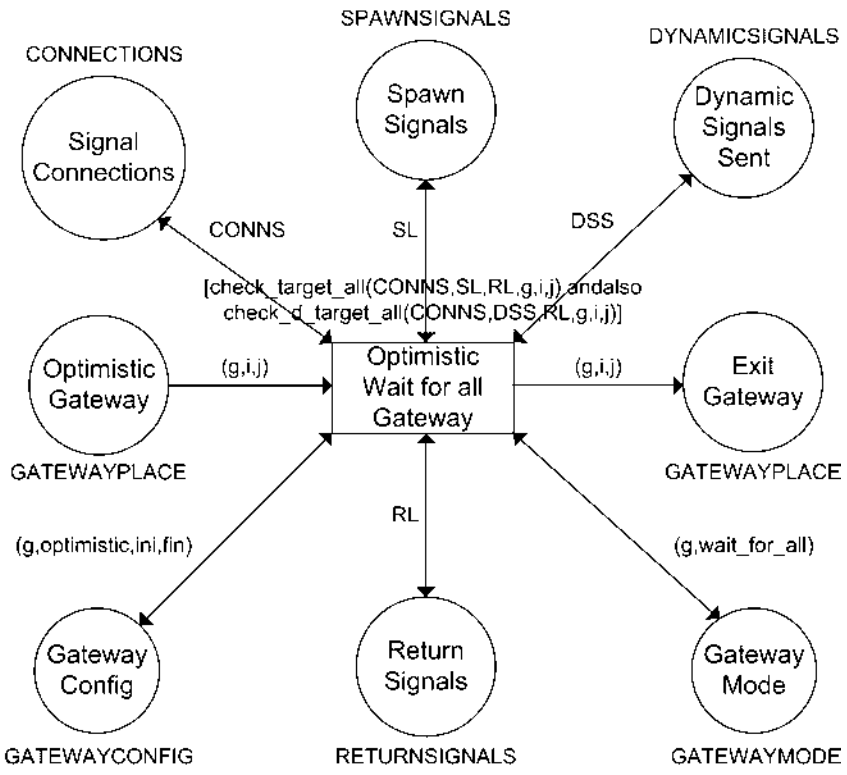


Figure 5.4: CPN - Receiving Signals at an Optimistic Wait-for-all Gateway

Alternatively, when a token representing an object instance is in the "Enter Gateway" place and the gateway is pessimistic, either the "Pessimistic Wait-for-one Gateway", "Pessimistic Wait-for-all Gateway" or "Pessimistic Wait-for-rel Gateway" transition is enabled. The choice of which transition is enabled depends on two things. Firstly, the gateway mode (wait-for-one, wait-for-all or wait-for-rel) ensures that only one transition will be enabled. Secondly, each transition has a guard condition that checks the number of signals that have been sent will only be enabled when the guard condition is satisfied. When fired, this transition moves the object instance token to the Pessimistic Gateway place.

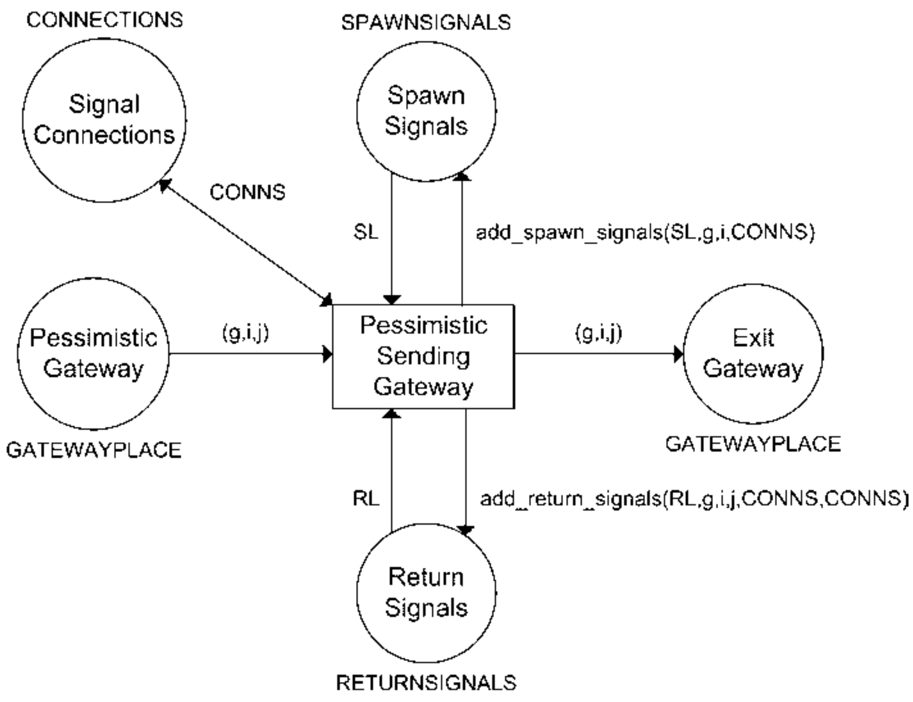


Figure 5.5: CPN - Pessimistic Gateway Sending

### 5.1.3 Processing Sub-State

After exiting the state pre-gateway, control-flow moves to the processing sub-state if the function *match_gateway(g,SGS)* evaluates to true where $g$ is the pre-gateway of the state that control-flow is being transferred to and $SGS$ is a list of state gateway tuples of the form *(g1,s,g2)* where $g1$ is the pre-gateway of $s$, $s$ is a processing sub-state and $g2$ is the post-gateway of $s$. The thread token then moves from the "Exit Gateway" place to the "Processing Sub-state" place. At this moment the tasks contained in the processing sub-state are enabled. As discussed in Section 3.1, control-flow is blocked in the processing sub-state until all mandatory tasks have completed. If *has_no_mand_tasks(s,TS)* or *not(check_mand_tasks_incomplete(s,TS,CT,TI))* evaluate to true, then either a state exists that has no mandatory tasks (this state can exit at any instant) or all mandatory tasks that belong to the processing sub-state have been completed. When the "From State To Gateway" transition fires, control-flow is passed from the processing sub-state to the state post-gateway. The relevant fragment of the CPN is shown in Figure 5.6.

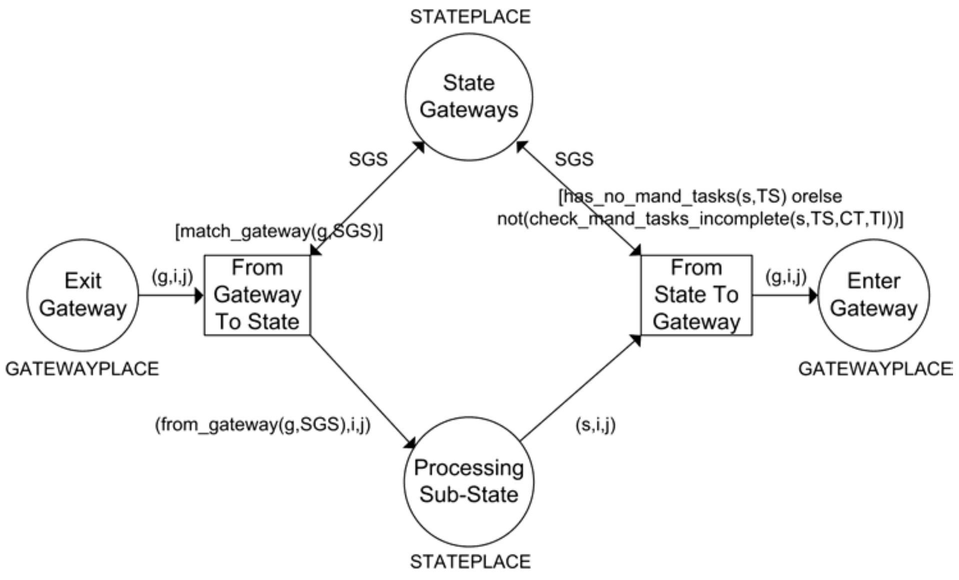


Figure 5.6: CPN - State

### 5.1.4 Tasks

When control-flow moves to the processing sub-state, instances of atomic tasks and/or multiple instance tasks that belong to that state can be created. The "Tasks" place contains a list of task/state mappings. A state can contain zero to many atomic tasks and/or multiple instance tasks. If the guard function *task_instance_missing(s,TI,TS)* evaluates to true where $s$ is a state, $TI$ is a list of active task instances and $TS$ is the list of tasks, the "Create task" transition is enabled, as shown in Figure 5.7. This function checks if there is a state in the "Processing Sub-State" place that contains a task that has not been created yet. Firing the "Create Task" transition adds an instance of that task to the "Created Tasks" place using the function *add_task_instance(s,TS,TI,i)*. Multiple instance tasks are handled slightly differently. If an instance of a multiple instance task has already been created then the tuple representing that task in the "Created Tasks" place is updated, meaning that the number of task instances recorded by the tuple is incremented. In addition, if the upper bound number of instances of a multiple instance task have already been created then no more instances of that task can be created.

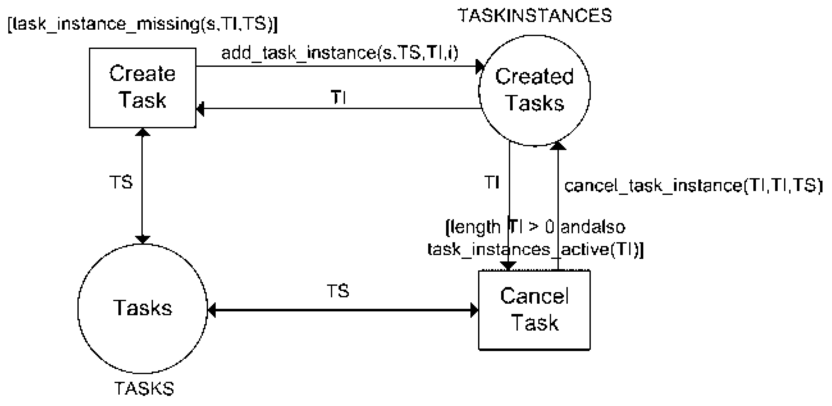

[task_instance_missing(s,TI,TS)]
TASKINSTANCES
add_task_instance(s,TS,TI,i)
Create Task
Created Tasks
TI
TS
TI
cancel_task_instance(TI,TI,TS)
[length TI > 0 andalso task_instances_active(TI)]
TS
Tasks
Cancel Task
TASKS

Figure 5.7: CPN - Creating and Cancelling Tasks

Active task instances can also be cancelled. A task instance may only be cancelled if there is one or more active tasks in the "Created Tasks" place. If this condition is satisfied, the "Cancel Task" transition can be fired and the function *cancel_task_instance(TI,TI,TS)* is executed. If an instance of a multiple instance task is cancelled, the number of active task instances is decremented for that task. If a task is cancelled, it can be created again. This is necessary since both 'optional' and 'mandatory' tasks can be cancelled. However, a state cannot exit until all mandatory tasks in that state have been completed. Thus the model allows both optional and mandatory tasks that have been cancelled to be *recreated*. Any previous information related to a cancelled instance of a task is not stored or recalled. In this way, cancellation of a mandatory task means that it is still possible to exit a state.

### 5.1.5 Completed Tasks

To complete an instance of a task, the "Execute Task" transition is fired. This transition is enabled if the *instance_available(TI)* function evaluates to true, which indicates that there is an "available" instance of a task that has been previously created in the "Created Tasks" place. When this transition is fired, the list of completed tasks $CT$ in the "Completed Tasks" place is updated to include the task that has just been completed by evaluating the *add_completed_task(CT,TI,TI)* function. The completion of tasks that belong to a state determines whether the state can exit. When all mandatory tasks that belong to a state have completed (*has_no_mand_tasks(s,TS)* is true), the "From State To Gateway" transition is enabled. Otherwise, if a state has no mandatory tasks then the state can exit at any instant (*not(check_mand_tasks_incomplete(s,TS,CT,TI))* is true). In the second case, if the state has optional tasks that have been created, they are left to complete but they have no effect on control-flow.

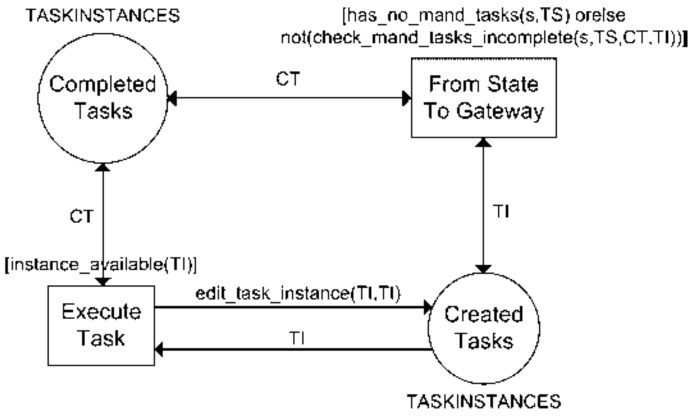


Figure 5.8: CPN - Completed Tasks

### 5.1.6 Spawn Signals

A spawn signal is a signal that creates an object instance. To send a spawn signal, both the "Optimistic Sending Gateway" and "Pessimistic Sending Gateway" transitions take the list of spawn signals from the "Spawn Signals" place and add any spawn signals that are sent from the gateway to the list of spawn signals using the function *add_spawn_signals(SL,g,i,CONNS)* where $SL$ is the list of spawn signals from the Spawn Signals place, $g$ is the name of the sending gateway, $i$ is the ID of the object thread and $CONNS$ is a list of signal connections from the "Signal Connections" place. A spawn signal is represented by a 4-tuple *(s,i,c,r)* where $s$ is the signal name, $i$ is the ID of the object that sent the signal, $c$ is the number of times that the spawn signal has been consumed and $r$ is the number of times that the spawn signal has been requested. A signal connection has an upper and lower bound. The number of spawn signals that are sent must be greater than or equal to the lower bound and less than or equal to the upper bound.

A spawn signal can be either "requested" or "consumed". When a number of spawn signals are sent by an object this number is recorded as both $r$, the number of "spawn requests" and $c$, the number of "spawn consumed". The object instances are "requested" meaning that object instances are not created at the same time they are requested. After a spawn request has been made, an object instance can then be created. This is handled by the "Spawn Child Instance" transition as shown in Figure 5.9. When this transition is fired, an instance of an object is created which is put in the "Enter Gateway" place and the number of spawn signals consumed is decremented. If $c$ for a spawn signal is zero, the "Spawn Child Instance" transition will not send any more signals of that spawn signal type.

### 5.1.7 Return Signals

When a spawn signal is added to the "Spawn Signals" place, one or more return signals are also added to the "Return Signals" place. This is done because the creation of an object instance with a spawn signal results in the creation of a new *relationship* between the parent object that sent the spawn signal and the child object that received it. A spawn signal sent from a parent to a child is always followed by a finish signal sent from the child and to the parent as well as zero or more message signals sent in either

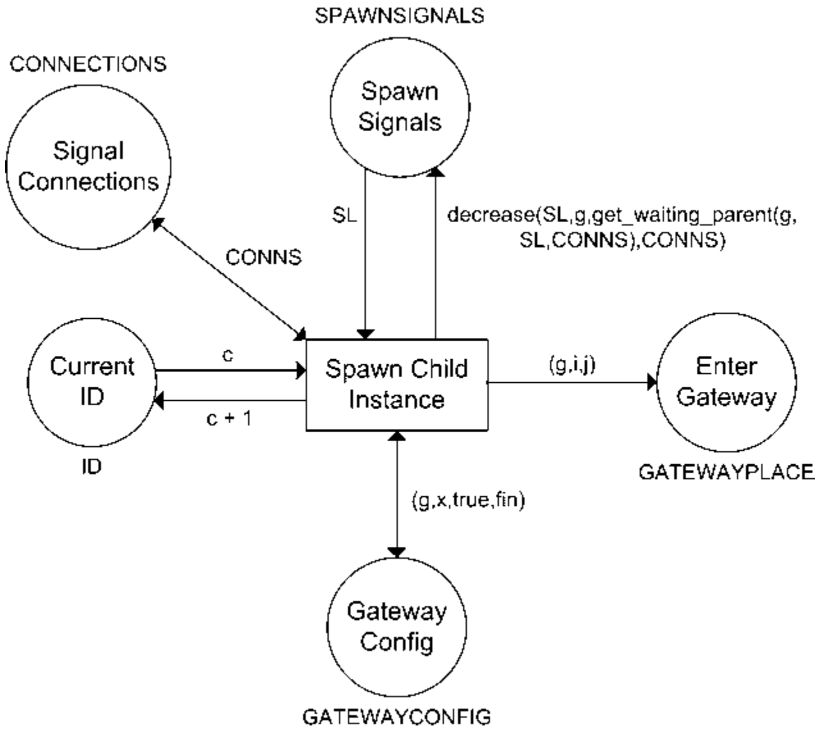


Figure 5.9: CPN - Child Instance Creation

directions. This necessitates creating a list of return signals to keep track of the number of each return signal that are "consumed". The number of return signals consumed is matched to the number of spawn signals sent, which is how control-flow synchronisation is performed at gateways depending on the gateway mode (wait-for-one, wait-for-rel or wait-for-all).

A return signal is represented by a 3-tuple *(s,i,c)* where $s$ is the signal name, $i$ is the object instance ID and $c$ is the number of return signals consumed. When a return signal is sent, $c$ is incremented. The return signals to be added to the "Return Signals" place are found by making a call to the function *add_return_signals(RL,g,i,j,CONNS,CONNS)* where $RL$ is a list of return signals, $g$ is a gateway name, $i$ is the parent ID, $j$ is the child ID and $CONNS$ is a list of signal connections. The list of signal connections in the "Signal Connections" place is parsed to evaluate which return signals (if any) will be added to the "Return Signals" place, which occurs as a result of sending a spawn signal.

A return signal connection is distinguishable from a spawn signal because the signal connection has a parent signal, which is defined as the spawn signal that created a relationship between two object types, whereas a spawn signal has no parent signal (i.e. *parent_signal(s)* $=$ *“”* if $s$ is a spawn signal, and $<>$ “” otherwise). If a signal connection has a parent signal and the source of the signal connection of that parent signal is the gateway that the thread token is currently in and the return signal does not already exist in the “Return Signals” place, then the signal is added to the “Return Signals” place. Otherwise, if $s$ is a return signal and the return signal already exists (meaning that the objects involved in the relationship that the return signal belongs to have already been created) no changes are made.

## 5.2 Extended Model in CPN

### 5.2.1 Creation Regions

When a thread token enters a gateway, the thread token can also enter a creation region. The list of creation regions is contained in the “Creation Regions” place. In the CPN a creation region is identified by the gateways that the creation region contains, so the “Creation Regions” place contains a list of 2-tuples of the form *(g,S)* where $g$ is a gateway name and $S$ is the set of signals that that can be sent from the creation region(s) that the gateway belongs to. If a gateway belongs to more than one creation region, the list of dynamic signals that can be sent from that gateway as a consequence of belonging to multiple creation regions are found in $S$.

As shown in Figure 5.10, when a transition between states is performed then the marking of the “Current Creation Regions” place is edited. This place maintains a list of “active” creation regions, which in turn determines the set of dynamic signals that can be sent at any particular moment in time. When a thread token enters a gateway, the creation regions that the gateway belongs to are added and the creation regions that are no longer active as a consequence of leaving the previous gateway are removed. This is done by calling the function *update_regions(g,T,CRS,ASIGS,i)* where $g$ is a gateway name, $T$ is a list of transitions, $CRS$ is a list of creation regions, $ASIGS$ is a list of currently active dynamic signals and $i$ is an object instance ID.

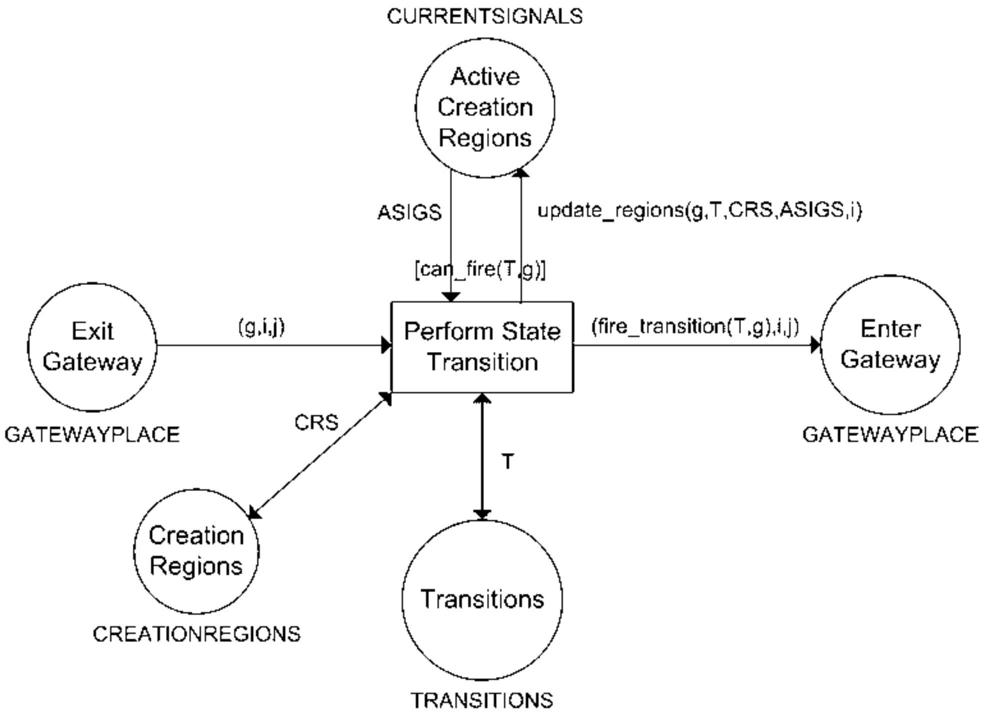


Figure 5.10: CPN - State Transition

If the post-gateway of the state from which the transition is being sent matches a gateway in the “Active Dynamic Signals” place then the list of signals corresponding to that gateway is removed from the place. If a match is found in the “Creation Regions” place for the pre-gateway of the state being entered, then the list of dynamic signals that can be sent by the creation region that the pre-gateway belongs to are added to the list of dynamic signals in the “Active Dynamic Signals” place.

If a thread token enters a gateway that is in a creation region then that creation region becomes an *active* creation region. This has the effect of allowing the object to send a set of dynamic signals that can be sent from the creation region. The types of dynamic signals that can be sent by a creation region are specified in the “Dynamic Connections” place. This place contains a list of dynamic connections that link dynamic signals to creation regions. If there is also a dynamic signal connection that can be sent from the current creation region, then the "Add Dynamic Signal" transition is enabled, as shown in Figure 5.11.

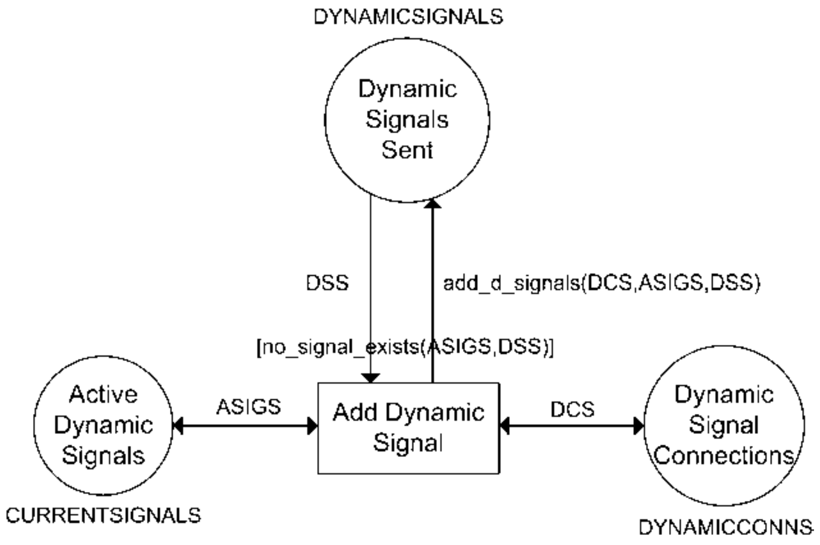


Figure 5.11: CPN - Add Dynamic Signal

A dynamic signal can only be added to the "Active Dynamic Signals" place if it does not already exist in that place. The transition guard function *no_signal_exists(ASIGS,DSS)* enforces this condition where $ASIGS$ is the list of dynamic signals that can be sent and $DSS$ is the list of currently active dynamic signals. If this function evaluates to true and the "Add Dynamic Signal" transition is fired, then the list of dynamic signals that can be sent are added to the "Dynamic Signals Sent" place by the function *add_d_signals(DCS,ASIGS,DSS)* where $DCS$ is a list of dynamic connections, $ASIGS$ is the list of dynamic signals that can be sent and $DSS$ is the list of currently active dynamic signals.

### 5.2.2 Creation, Referral and Nesting Signals

The creation, referral and nesting signals are all sent by the "Create" transition in the CPN as shown in Figure 5.12. To understand why this occurs we need to explain the design decision that allows these three signals to be grouped together. The object designation information (COROB, JOB and ROB) that specifies the source and target object types of particular signal types is abstracted away in the CPN (e.g. a nesting signal is sent from a ROB to a COROB). Making a differentiation between the object designations is of most use to the model designer at design time since the object designations provide assistance to develop a syntactically correct object model. At the implementation phase, it is no longer of any relevance to recall an object designation because the creation, referral and nesting signals essentially all *create* a new object instance. For this reason, these three dynamic signals share the "Create" transition in the CPN.

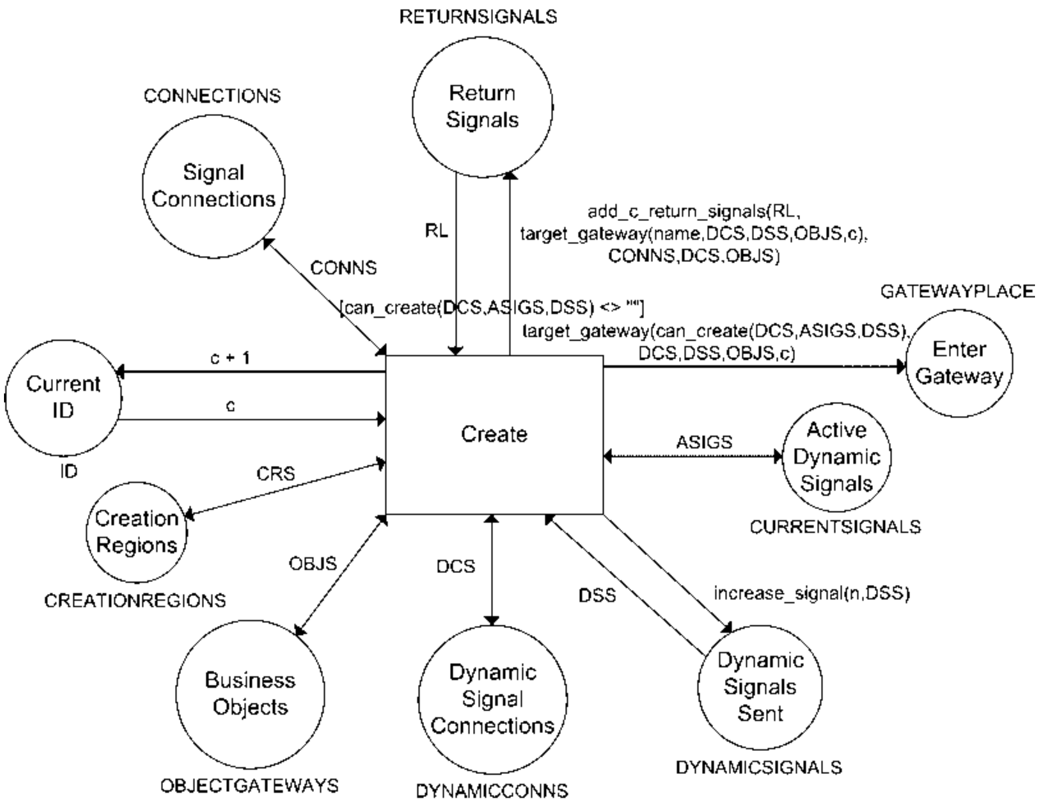


Figure 5.12: CPN - Creation

This method of object instance creation relies upon the specification of object inheritance associations at design time. The benefit of this approach is two-fold: (1) it has the effect of grouping related objects together which promotes structured object model design by making it necessary to create inheritance associations between objects that may be created by a particular dynamic signal, and (2) it enables runtime type flexibility since the type of object that will be created by the dynamic signal is deferred until runtime. An instance of an object can be created if a dynamic signal of type *creation*, *referral* or *nesting* is in the "Dynamic Signals Sent" place and the number of dynamic signals sent is less than the upper bound specified for that signal. The guard transition function *can_create(DCS,ASIGS,DSS)* enforces this restriction where $DCS$ is a list of dynamic connections, $ASIGS$ is a list of active dynamic signals and $DSS$ is a list of dynamic signals sent. When fired, the transition increments the number of dynamic signals sent, adds a new instance of the target object to the Enter Gateway place, increments the Instance ID and adds any return signals for this object to the "Return Signals" place.

### 5.2.3 Delegation Signals

Delegation is a different operation because it involves more than creating an instance. Delegation changes the type of an executing instance of an object to another type. This necessitates a different transition in the CPN to capture this fundamental difference. Other concerns also need to be considered by delegation. A restriction is placed on when delegation is performed that confines the ability of an object to delegate only when a thread token is in the processing sub-state. This is a design decision that was made to simplify the complexity of performing delegation. It also means that the CPN matches how delegation is performed by a user, which is done when an object instance is in a particular state, not a gateway.

If the function *can_delegate(DCS, ASIGS,DSS,s,CRS)* evaluates to a non-empty string, delegation can only be performed. The function tests that the current state belongs to a creation region that sends a delegation signal. If true, the name of the delegation signal is returned. The name of the signal is passed to the function *delegatee(n,DCS,DSS,OBJS,c,j)* which returns the initial gateway of the delegatee. The thread token that represents the delegator is removed from the "Processing Sub-state" place, which is converted into a token that represents the delegatee using the function *target_gateway(n,DCS,DSS,OBJS,c)* and is placed in the "Enter Gateway" place.

Lastly, it is a condition of the delegator that the delegatee must "take over" as the sender and receiver of signals that belong to the delegator which are found in the "Return Signals" place. The function *edit_d_return_signals(RL,g,CONNS,DCS,OBJS,i,c)* inspects each return signal for the ID of the delegator and updates it to the ID of the delegatee. As can be observed in Figure 5.13, delegation of an object lifecycle involves evaluating a significant number of places in the CPN. This is due to the large number of dependencies involved. After each of the functions related to delegation have been evaluated, the object lifecycle of the delegatee can begin.

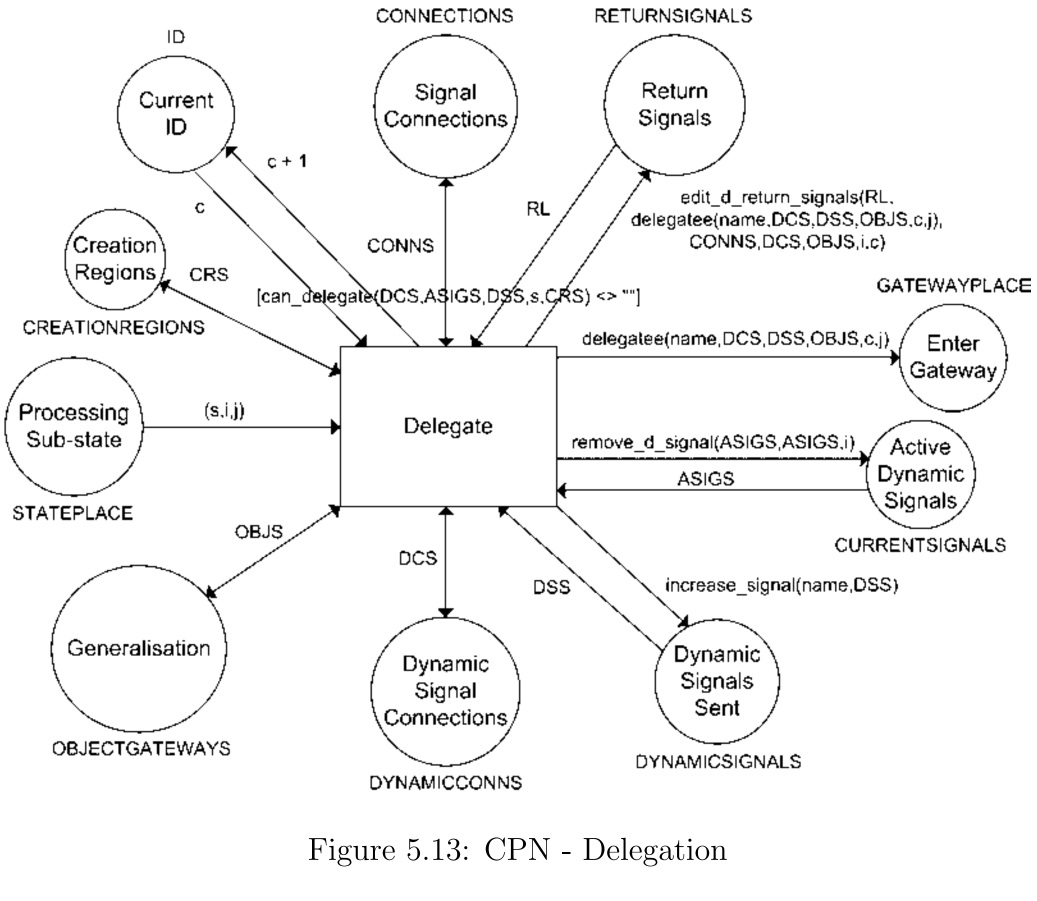


Figure 5.13: CPN - Delegation

As shown in Figure 5.13, delegation may only be performed when a state machine lifecycle is in a state. If delegation could be performed from a gateway this would potentially impact the consistency of an executing model. For example, if delegation was performed when an instance of an object was in a pessimistic gateway that had received signals but had not yet sent signals, this could have the impact of causing control-flow synchronisation problems because another object may be expecting to receive signals that will never be sent. The possibility that such scenarios can occur should be taken into account when designing flexible process models.

Delegation is a powerful tool for a process modeller since it allows a potentially large number of alternative paths in a process model to be specified at design time, while maintaining process control. This is beneficial in circumstances when it not known with a great amount of certainty which *type* of object is needed to complete a task until a late stage in process execution.

## 5.3 Summary and Discussion

The CPN described in this chapter is a formal definition of the behavioural semantics of both the base object-centric meta-model (presented in Chapter 3) and extended meta-model (presented in Chapter 4), which was implemented and tested using CPN Tools. Development of the CPN was completed in two stages, the base meta-model concepts were designed and implemented in the first stage, then the extended meta-model concepts were added in the second stage. The addition of the extended meta-model concepts in the second stage required only minimal changes to the base meta-model. As a result of this approach to CPN development, the CPN can be divided into two sections that contains either the base meta-model concepts or the extended meta-model concepts. These sections can then be further broken down into the individual parts, e.g. states, transitions, gateways and static signals for the base meta-model, and creation regions and dynamic signals for the extended meta-model.

The CPN also allows us to test the behaviour of object-centric models using an export function in the FlexConnect modelling tool. Using this approach to testing object-oriented process models there is no need to create a new CPN or remodel an existing CPN each time that a different O-C process model is tested, since the CPN is based on the FlexConnect meta-model. This means that it is capable of executing any model created by the FlexConnect modelling tool. This approach is appealing since CPN development can be quite difficult and time-consuming. However, an existing limitation in the CPN is that object models loaded into the CPN are assumed to be syntactically correct, because no syntax checks are performed other than CPN-ML syntax and type checks performed by CPN Tools.

# Chapter 6

# FlexConnect Pattern Evaluation

In this chapter we examine the capabilities and distinctive features of the FlexConnect modelling language. This has been done by evaluating the language using the catalogue of Revised Workflow Control-flow Patterns [83] and the Taxonomy of Flexibility [89]. The evaluations enable us to reason about the *suitability* of an object-centric approach to process modelling. A pattern-based evaluation of the *flexible* aspects of FlexConnect and YAWL (including the YAWL Worklet service) is also presented to compare and contrast how these languages support various types of ad-hoc variations. The control-flow pattern evaluation of FlexConnect in this chapter is an extension of the report available at [72].

## 6.1 Control-flow Pattern Evaluation

In this section the control-flow constructs of FlexConnect are evaluated. The control-flow pattern evaluation is presented with a description and model fragments (where relevant), to show how support for each pattern is achieved by FlexConnect.

### 6.1.1 Basic Control Patterns

#### WCP-1 Sequence

An activity in a workflow process is enabled after the completion of a preceding activity in the same process.

In FlexConnect, tasks are contained within states. The completion of each task is either compulsory or optional. A compulsory task must be completed before a state can be exited, whereas an optional task has no effect on control-flow whatsoever. In addition, a state can contain any number of compulsory and optional tasks. The completion of all compulsory tasks in a state means that the state can exit. A transition can then be taken from that state to another state in the same state machine. In Figure 6.1 we show full support for the Sequence pattern by a sequence of states from the Initial State, to State 001, to State 002, to the Final State.

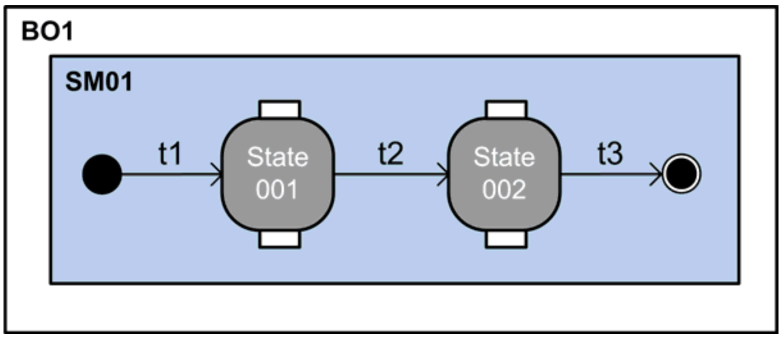


Figure 6.1: WCP-1 – Sequence

#### WCP-2 Parallel Split (AND-Split)

A branch diverges into two or more parallel branches which each execute concurrently.

To support the AND-Split in FlexConnect, it is necessary to divide the model into several state machines that will execute independently. Specifically, an AND-Split is captured by a parent state machine that instantiates two or more child state machine instances. An instance of a child state machine is created by a "spawn signal". A spawn signal is sent from either a pre- or post-gateway of a state to the initial state of another state machine. The child state machine lifecycle executes in parallel to its parent. As shown in Figure 6.2, the AND-Split is fully supported by sending two spawn signals (Sig01 and Sig02) with multiplicity of *1..1* from the pre-gateway of State 001, which will split control-flow into two parallel branches, namely the child state machines SM02 and SM03.

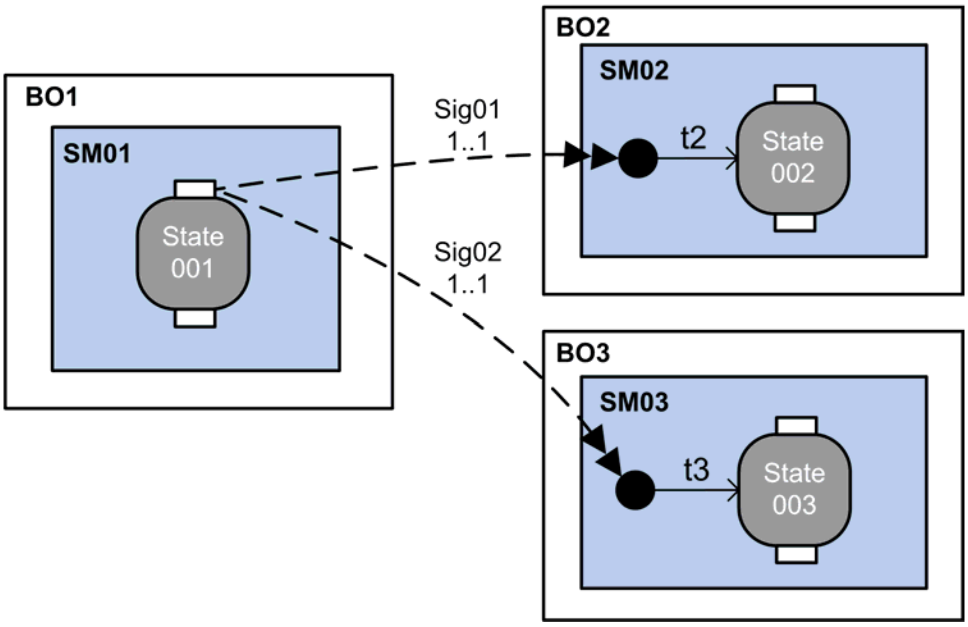


Figure 6.2: WCP-2 – AND-Split

### WCP-3 Synchronisation (AND-Join)

| Two or more branches converge into a single subsequent branch. Control passes to a single subsequent branch after all input branches have completed. |
|---|

The AND-Join is fully supported in FlexConnect by a gateway that is the target of two or more signals. In Figure 6.3, the child state machines SM02 and SM03 send finish signals to the post-gateway of State 004 in the parent state machine SM01. The gateway mode property is "wait-for-all", which means that the state machine lifecycle will wait (or block) until all signals that will be sent to the gateway are received. An interesting addition to the support for this pattern is due to the parent state machine maintaining a list of active children. Using this list the parent can query the number and type of its active children. This allows the parent to avoid the possibility of deadlock at an AND-Join in case any (or all) of the child state machines that the parent expects to receive a signal from are cancelled.

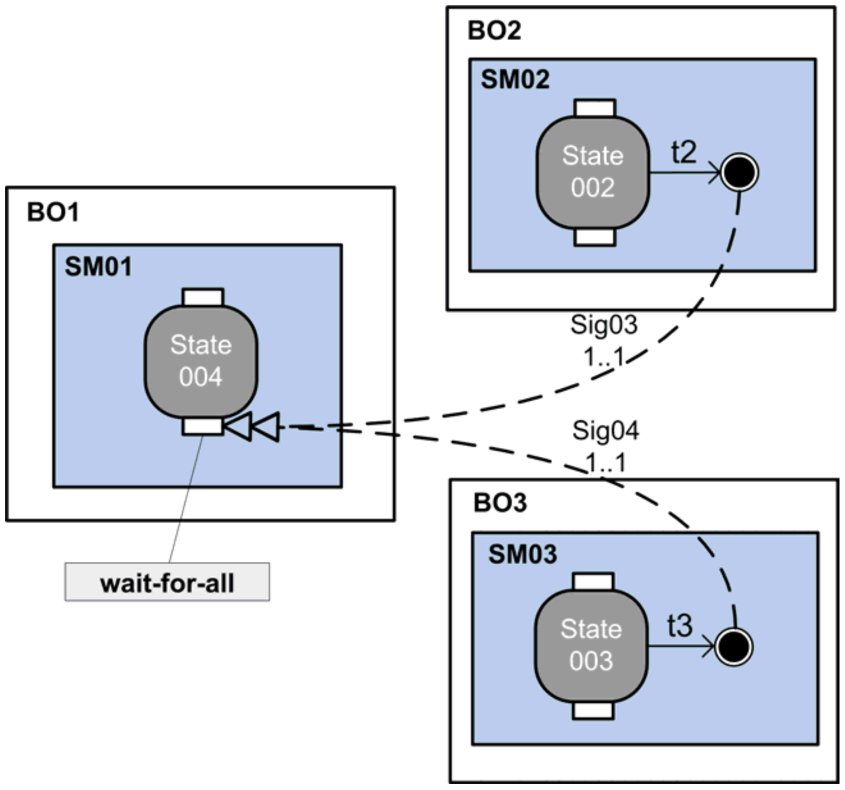


Figure 6.3: WCP-3 – AND-Join

### WCP-4 Exclusive Choice (XOR-Split)

A branch diverges into two or more branches. The thread of control is passed to one of the outgoing branches based upon a logical expression associated with the branch.

A transition can have an optional condition. To capture an XOR-Split, each outgoing transition from a state has a condition specified (except the default transition), which are evaluated after the state has exited. Based on the condition evaluation, a transition is performed to the state which corresponds to the first condition that is satisfied. The default transition is evaluated last. A limitation is that an exclusive choice can only be specified between states that exist in the same state machine, due to the fact that conditions are only attached to transitions and are not attached to signals. Conditions are shown in Figure 6.4 as part of an ECA rule on each transition.

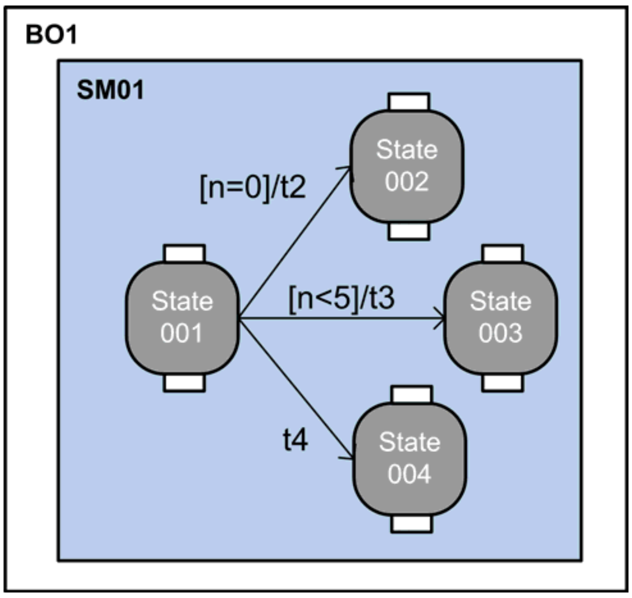


Figure 6.4: WCP-4 – Exclusive choice in the same state machine

## WCP-5 Simple Merge (XOR-Join)

Two or more branches converge into a single branch. Each incoming branch passes a thread of control to the subsequent branch.

The simple merge is supported by FlexConnect in two ways; within the same state machine and across different state machines. Within the same state machine, the simple merge is captured by a state that has multiple incoming transitions, as shown in Figure 6.5. When one of the transitions attached to State 005 is performed, the simple merge occurs. An instance of a state machine can be in only one state at any point in time, guaranteeing that only one incoming transition out-of-many will be performed to enter a state. The simple merge may also be achieved across different state machines that communicate by transmitting signals, as shown in Figure 6.6. The mode of the gateway that represents the merge is specified as “wait-for-one”. This gateway mode means that control-flow is released by the gateway when the first incoming signal is received, but also has the consequence that the gateway must receive at least one signal or control-flow will not be released.

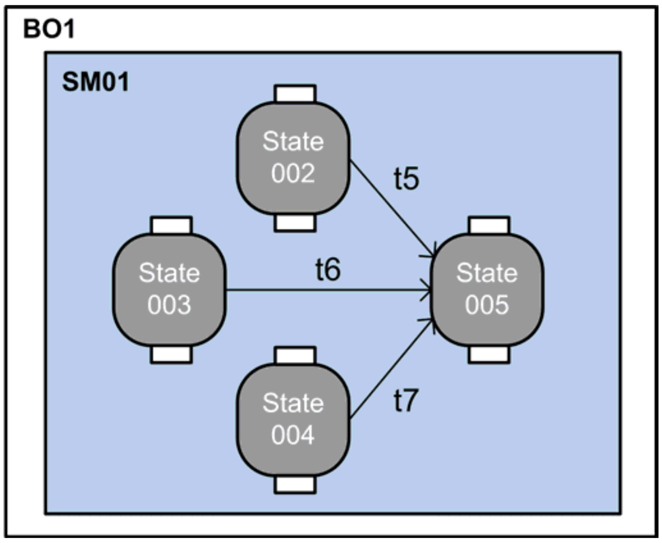


Figure 6.5: WCP-5 – Simple merge within the same state machine

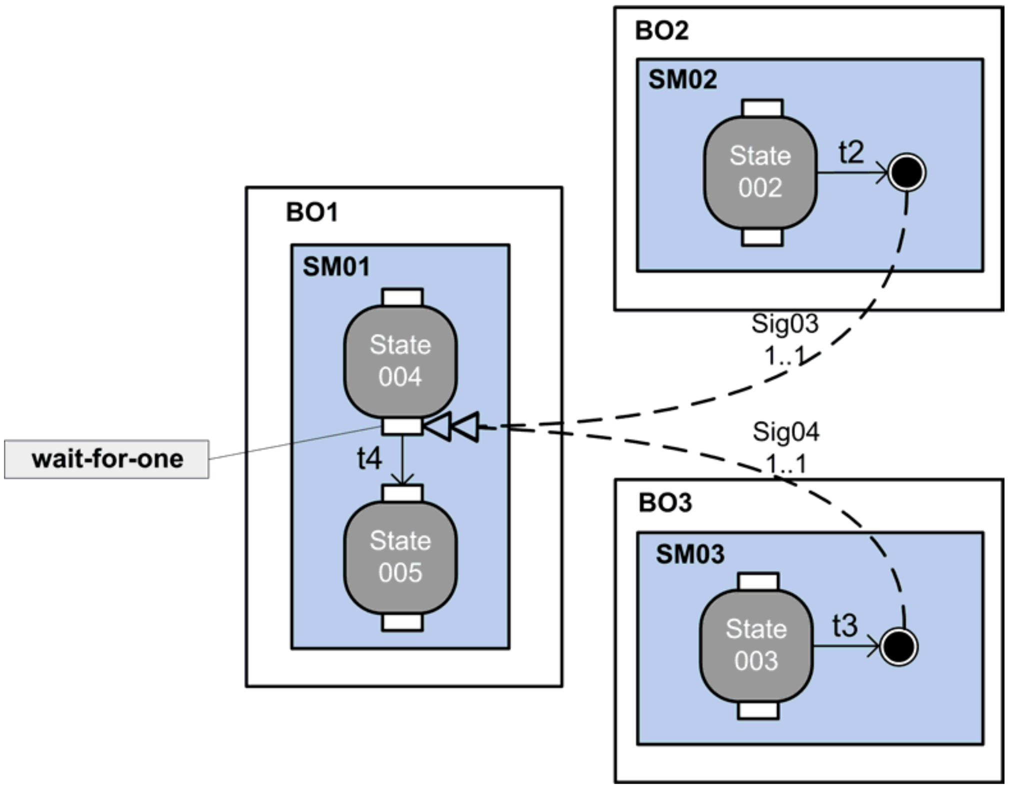


Figure 6.6: WCP-5 – Simple merge across different state machines

## 6.1.2 Advanced Branching Patterns

### WCP-6 Multi-choice (OR-Split)

| |
|---|
| A branch diverges into two or more branches. The thread of control is passed to one or more of the outgoing branches based upon the result of logical expressions associated with each branch. |

In FlexConnect, the OR-Split is captured by a gateway that sends two or more signals that have a lower bound multiplicity of *0*, as shown in Figure 6.7. However, instead of basing the result of the split on the evaluation of logical expressions, the lower and upper bounds of the outgoing signal multiplicity are interpreted. The lower bound multiplicity of *0* of the outgoing signals from the pre-gateway of State 001 means that multiple choices can be made from this gateway. In the example in Figure 6.7, four choices can occur as the outcome of the split: 1) An instance of SM02 is created, 2) An instance of SM03 is created, 3) An instance of SM02 and SM03 is created, or 4) No instances of either SM02 or SM03 are created.

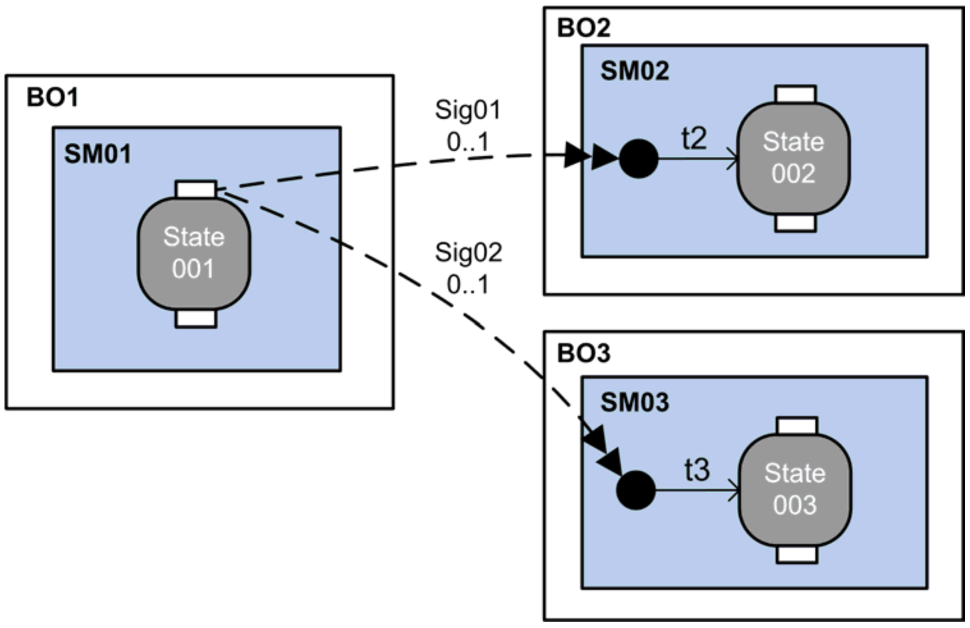


Figure 6.7: WCP-6 – Multi-choice

### WCP-7 Structured Synchronising Merge

Provides a means of synchronising all splits previously made from a multi-choice. Four context conditions exist. If only one path, alternative branches should converge without synchronisation, otherwise all active branches from a multi-choice split must synchronise.

To support this pattern, the post-gateway of State 002 in state machine SM01 is set to "wait-for-all", which means that a state in the parent state machine will wait until all signals from all children have been received (where 'all' can also mean 'none' if no child instances have been created). In addition, a parent state machine can query the number and type of its children that are active. This enables the parent to only wait for active children since it knows the number of signals it can expect to receive from the children previously created by an OR-Split (the pre-gateway of State 001). For example, as shown in Figure 6.8 the parent SM01 will exit from the post-gateway of State 002 when all finish signals (Sig03 and Sig04) have been received. The upper bound of these signals is unbounded, but since SM01 knows the number of active children, it also knows the number of expected signals to receive from instances of SM02 and SM03.

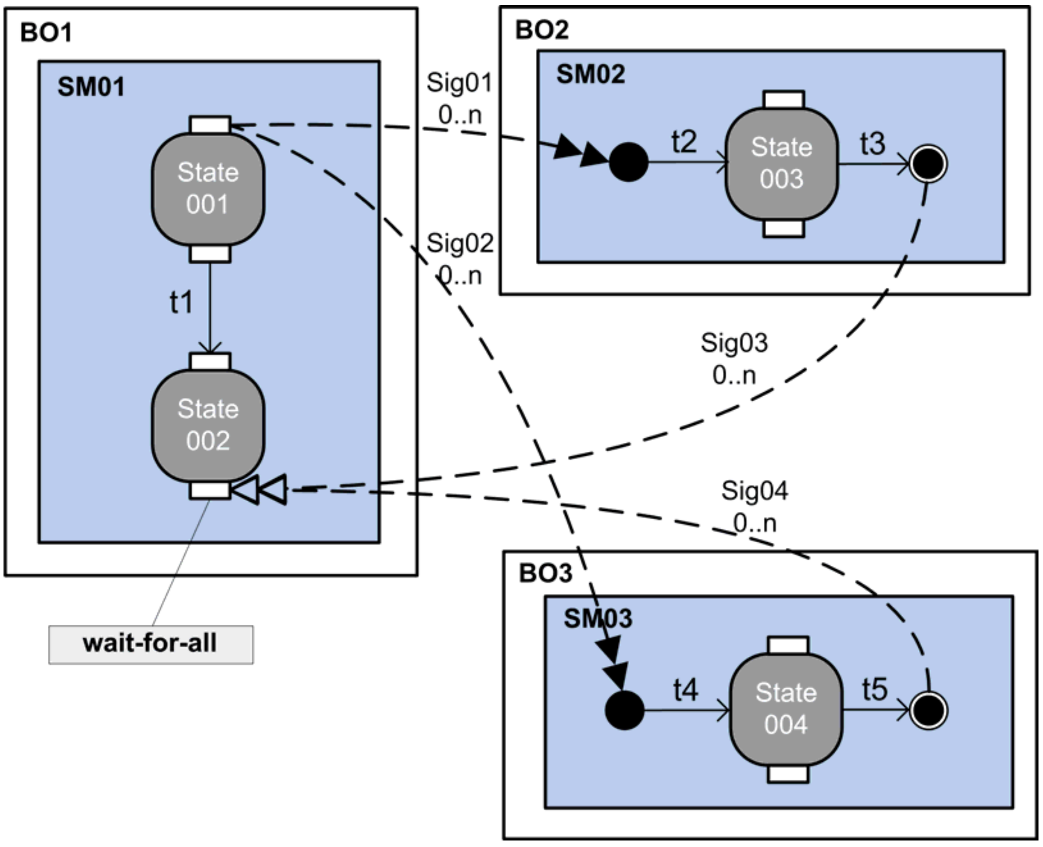


Figure 6.8: WCP-7 – Structured Synchronising Merge

### WCP-8 Multi-Merge

| |
|---|
| Every active path into the merge triggers the task after the merge. No synchronisation occurs at the join. |

When the state machines SM02 and SM03 enter State 002 and State 003 respectively, a multi-merge will occur since each state machine creates a new instance of state machine SM04 with the spawn signals Sig03 or Sig04. This means that State 004 is entered twice with no synchronisation occurring between the state machines with respect to State 002 and State 003, as shown in Figure 6.9. This satisfies the conditions for this pattern that "every active path into the merge triggers the task after the merge" and that "no synchronisation occurs at the join".

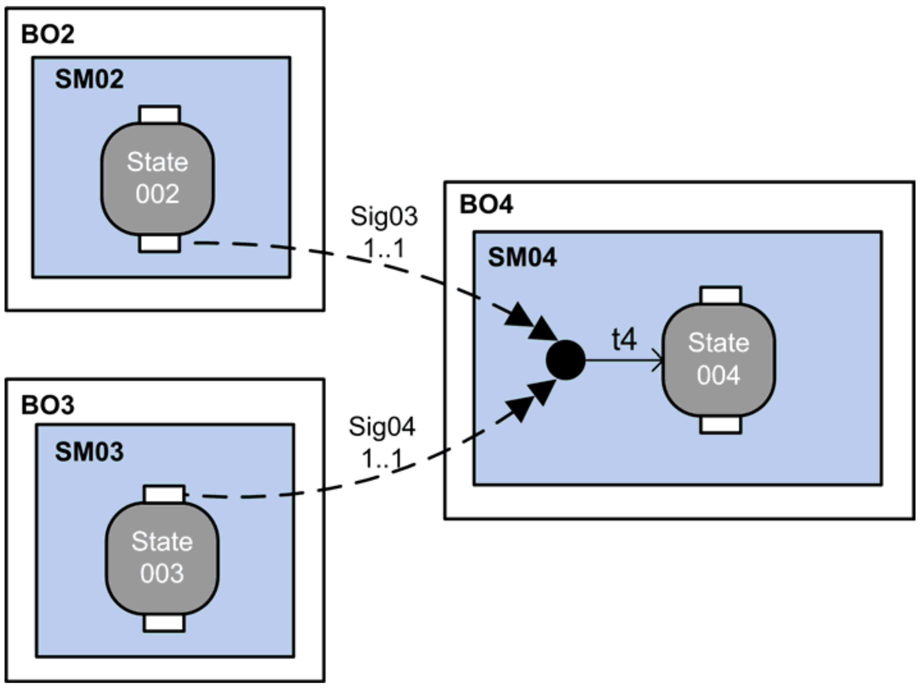


Figure 6.9: WCP-8 – Multi-Merge

## WCP-9 Structured Discriminator

| |
|---|
| Two or more branches converge into a single subsequent branch following an earlier divergence in the process model. The thread of control is passed on from the first active incoming branch. Subsequent active incoming branches are not passed on. The discriminator resets once all incoming branches have been enabled. |

The structured discriminator is implemented by a waiting state that contains the merge point. As shown in Figure 6.10, each child state machine (SM02 and SM03) sends a finish signal to a post-gateway in the parent (SM01). The post-gateway mode is “wait-for-one”, meaning that the merge is performed when the first child signal received. If multiple signals arrive simultaneously, the choice of which signal performs the merge is nondeterministic. The construct resets after all signals have been received.

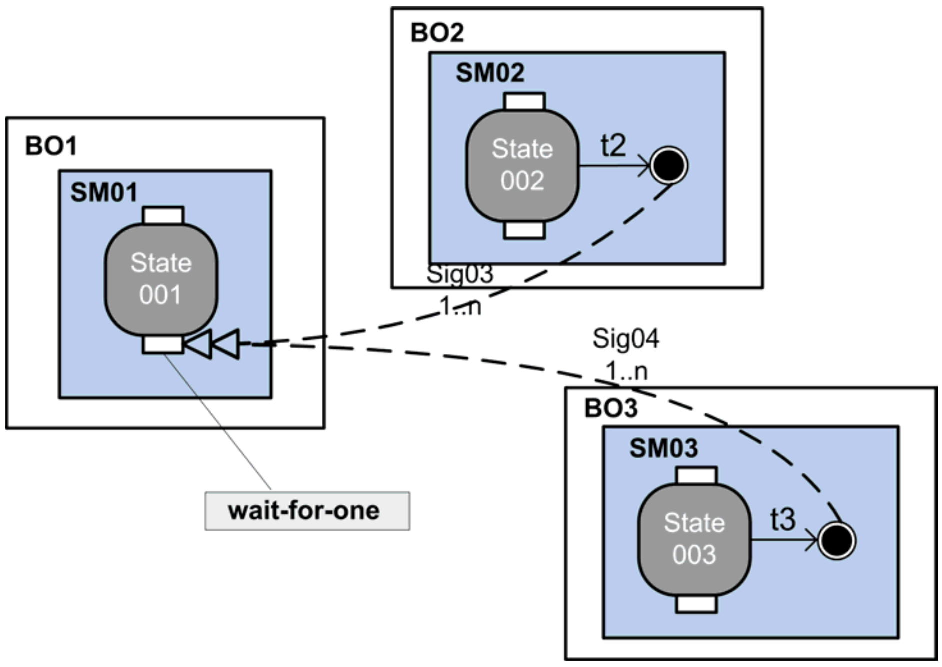


Figure 6.10: WCP-9 – Structured Discriminator

### 6.1.3 Structural Patterns

#### WCP-10 Arbitrary Cycles (Unstructured loop)

| The ability to represent cycles in a process model that have more than one entry or exit point. |
|---|

FlexConnect does not explicitly define loops. This is because the state machine paradigm that FlexConnect is based upon does not treat unstructured loops as a distinct modelling construct, since any state in a state machine (apart from the initial and final states) can have more than one incoming or outgoing transition. An unstructured loop consists of a repeating sequence of states and transitions as shown in Figure 6.11. An unstructured loop in this example begins at State 001 to State 002 to State 003 to State 004 and loops back to State 001. In this loop there are two exit points, from either State 003 or State 004 to the final state, meaning that the loop can be classified as an arbitrary cycle.

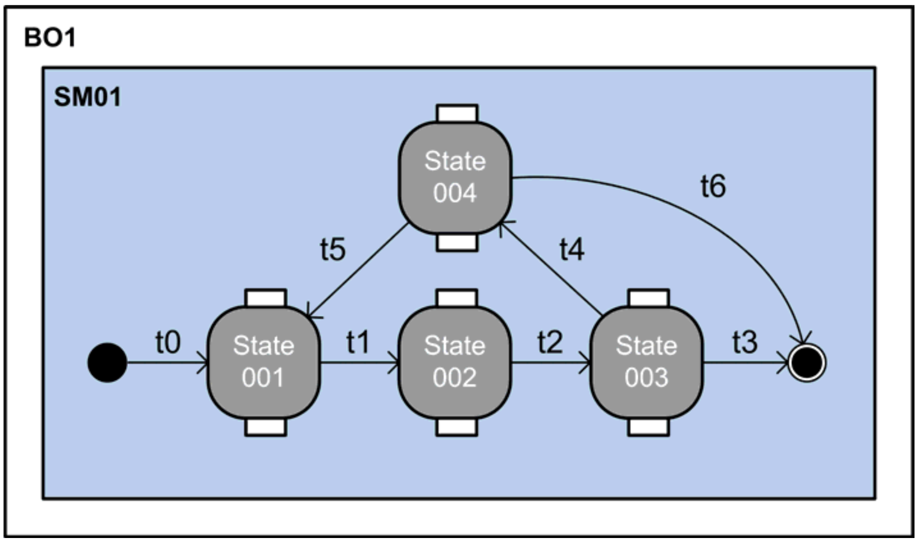


Figure 6.11: WCP-10 – Arbitrary Cycles

### WCP-11 Implicit Termination

A given process or sub-process should terminate when there are no more remaining work items that can be done now or at any time in the future.

Implicit termination is not supported by FlexConnect because a process instance is only terminated when the final state is reached (which is an example of explicit termination).

### 6.1.4 Multiple Instance Patterns

#### WCP-12 Multiple Instances without Synchronisation

Within a given process, multiple instances of an activity can be created. These instances are independent of each other and run concurrently. There is no need to synchronise them upon completion.

There are two types of multiple instances in FlexConnect. Firstly, there are multiple instance tasks that are found within a state. The completion of multiple instance tasks is specified as either ‘compulsory’ or ‘optional’, meaning that the completion of the task instances are either required or not required in order to exit the state that the multiple instance task belongs to. An optional multiple instance task is started and is left to complete without synchronising, meaning that the state does not wait for the completion of optional tasks, which satisfies the criteria for no synchronisation upon completion. Secondly, multiple instances of child state machines can be created. A child state machine can be seen as a composite task that contains more than one task, executed in a sub-process. However, a child state machine does not always return a finish signal, as shown in Figure 6.12.

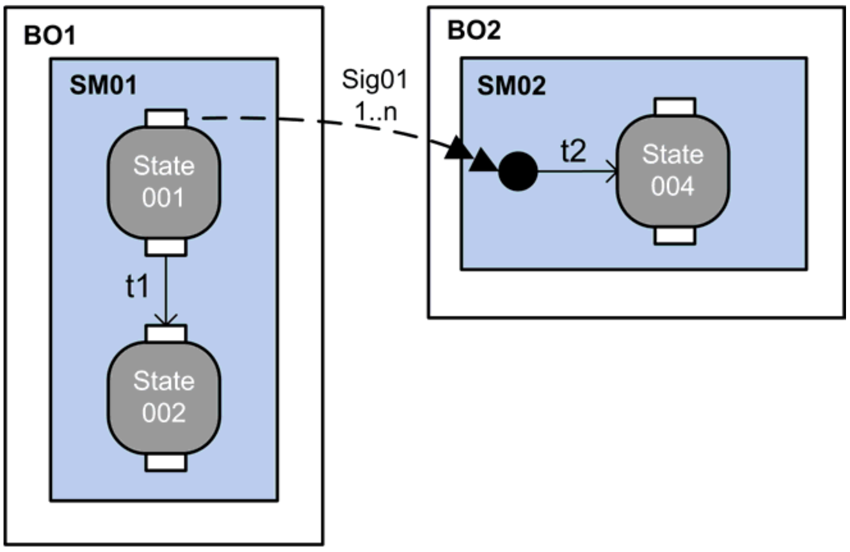


Figure 6.12: WCP-12 – MI without synchronisation

### WCP-13 Multiple Instances with a priori Design Knowledge

Within a given process, multiple instances of an activity can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronise them upon completion before any subsequent tasks can be triggered.

Similar to support for the WCP-12 Multiple Instances without synchronisation pattern, FlexConnect supports this pattern with both multiple instance tasks in states and multiple instances of state machines. To support this pattern using multiple instance tasks, a multiple instance task is defined in a state and its completion is set to 'compulsory'. This means that the state will block until all instances of the task are complete at which time synchronisation will occur and the state will exit. To support the pattern using state machines, a spawn signal is used to create child state machine instances, and each child state machine returns a finish signal to the parent at lifecycle conclusion. The mode of the gateway in the parent state machine that receives finish signals from the children is set to "wait-for-all", which blocks until a finish signal has been received from all child instances, as shown in Figure 6.13.

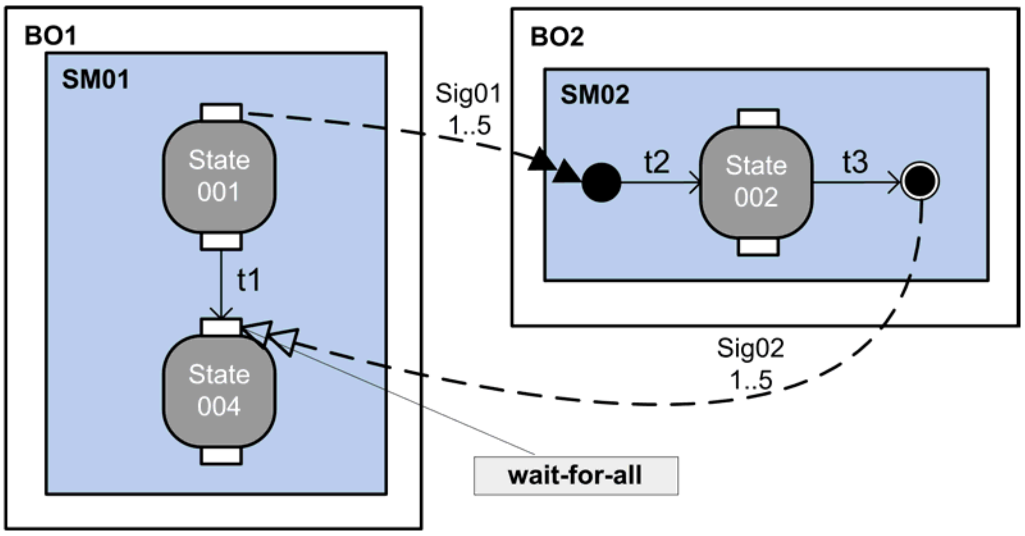


Figure 6.13: WCP-13 – MI with a priori design knowledge

## WCP-14 Multiple Instances with a priori Runtime Knowledge

Within a given process, multiple instances of an activity can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but is known before the activity instances must be created. Once initiated, these instances are independent of each other and run concurrently. It is necessary to synchronise them upon completion before any subsequent tasks can be triggered.

Support for this pattern is achieved in a similar way to the pattern WCP-13 Multiple Instances with a priori Design Knowledge. At design time in this case, the upper bound of the multiple instance task or a spawn signal is specified as unbounded ($n$), which allows the number of instances needed at runtime to be created as needed, as shown in Figure 6.14. This allows any number of instances to be created which may be dependent on state data or resource availability.

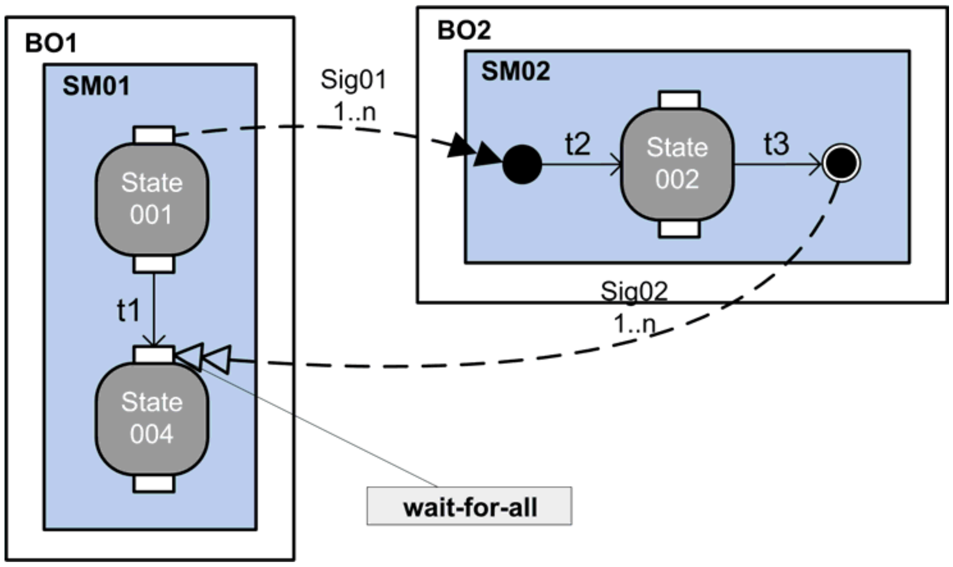


Figure 6.14: WCP-14 – MI with a priori runtime knowledge

### WCP-15 Multiple Instances without a priori Runtime Knowledge

Within a given process instance, multiple instances of an activity can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but is known before the activity instances must be created. At any time, whilst instances are running, it is possible for additional instances to be created. It is necessary to synchronise them upon completion.

Support for this pattern is achieved in much the same manner as the pattern WCP-14 Multiple Instances with a priori Design Knowledge. At runtime, a number of instances are created as needed due to the unbounded ($n$) upper bound of the spawn signal, as shown in Figure 6.15. This allows any number of instances to be created which may be dependent on state data or resource availability that is specified at runtime. Synchronisation of all created instances of SM02 is performed at the pre-gateway of State 004, which waits for all created instances of SM02 to complete.

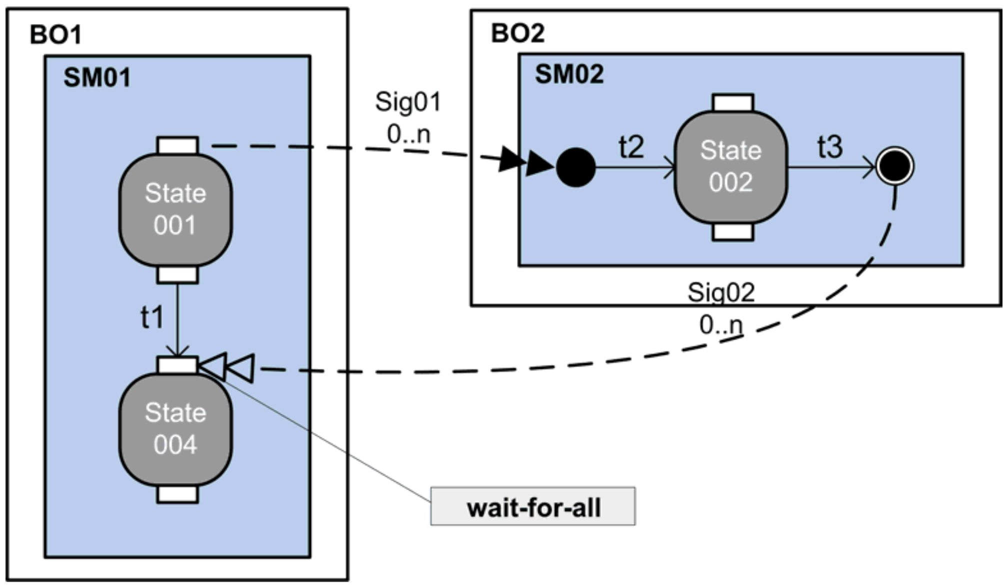


Figure 6.15: WCP-15 – MI without prior runtime knowledge

### 6.1.5 State-Based Patterns

#### WCP-16 Deferred Choice (Deferred XOR-Split)

| |
|---|
| One of several branches is chosen based upon interaction with the environment. Prior to the decision all possible future courses of execution are presented. After the decision is made execution alternatives other than the one selected are removed. |

In FlexConnect a choice between multiple transitions emanating from a given state is always deferred until the occurrence of an event that triggers the exit from a state. The example shown in Figure 6.16 shows a deferred choice where three transitions are possible from a single state and the decision of which transition to take is not made until State 001 has exited.

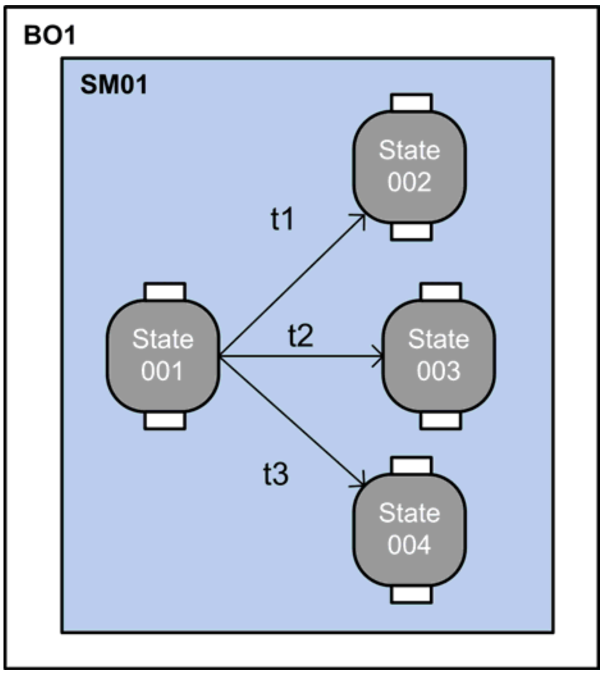


Figure 6.16: WCP-16 – Deferred choice between three states.

### WCP-17 Interleaved Parallel Routing

A set of tasks is executed in a sequence that is determined at runtime. No two activities can be active at the same time. The tasks synchronise upon completion.

The interleaved parallel routing pattern is not supported because Flex-Connect has no notion of a semaphore. That is, there is no construct that can be used to restrict task execution across object lifecycles such that no more than one instance of a task can be executing from a set of tasks in the manner required to support interleaved parallel routing.

### WCP-18 Milestone

An activity is enabled, but cannot be completed until a milestone is reached elsewhere, which has not expired.

In the example shown in Figure 6.17, there is a milestone at the post-gateway of State 002 in SM01. At the pre-gateway of State 001, an instance of SM02 is created and either zero or one instance of SM03 is created. If an instance of SM03 is created, this impacts the blocking behaviour of the post-gateway of State 002. A transition to State 003 cannot be performed until the post-gateway of State 002 has received a finish signal from SM02 and a message signal from the post-gateway of State 005, depending on whether an instance of SM03 was created. The post-gateway of State 002 will only wait to receive a signal from State 005 if an instance of SM03 exists, due to the “wait-for-all” mode of this gateway. If an instance of SM03 exists, State 003 cannot be exited until State 005 in SM03 has exited.

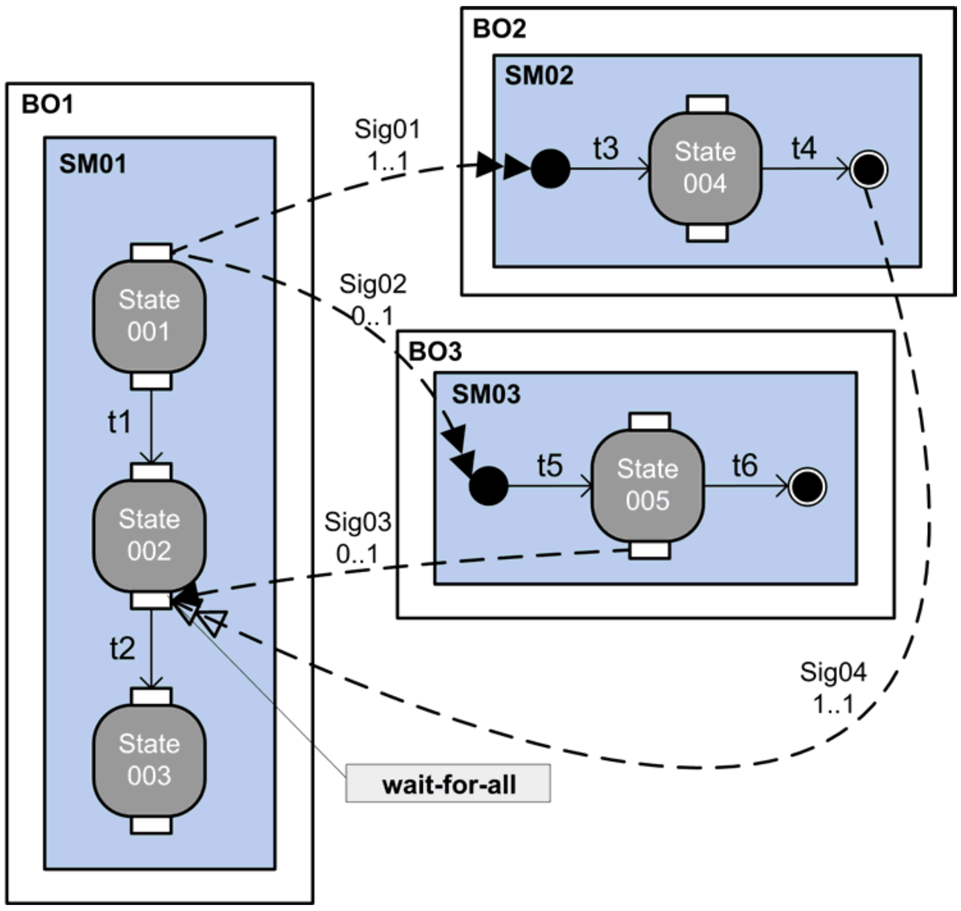


Figure 6.17: WCP-18 – Milestone

### 6.1.6 Cancellation Patterns

#### WCP-19 Cancel Activity

| Disable an enabled activity, before it is completed. |
|---|

Tasks may be cancelled at runtime by explicitly cancelling a running instance of a task in FlexConnect model. Both optional and compulsory tasks can be cancelled. A task instance that has been cancelled can be restarted.

#### WCP-20 Cancel Case

A workflow instance (a case) and its descendants are disabled and removed.

On the surface cancel case looks relatively easy to support, but it is not supported in FlexConnect. There is considerable difficulty in supporting cancel case in FlexConnect since this kind of cancellation has wide ramifications. Let us discuss this pattern in terms of the CPN from Chapter 5. If a parent is cancelled, its children should also be cancelled, and those children may be parents to additional children which also need to be cancelled. The token representing each object lifecycle may be in one of three places; the pre-gateway, processing sub-state or post-gateway. Also, these objects may or may not have sent static signals and/or dynamic signals that need to be removed and may or may not be in a creation region that needs to be removed. Owing to the design of FlexConnect this pattern is too cumbersome to support. However, single cases (state machine instances) can be cancelled as required on an individual basis.

### 6.1.7 Extended Workflow Control Flow Patterns

#### WCP-21 Structured Loop

The ability to execute an activity or sub-process repeatedly. The loop has either a pre or post-condition that determines whether it can continue. The looping structure has a single entry and exit point.

This pattern is not supported by FlexConnect. The main problem is that the start and end of a loop cannot be explicitly defined as a loop entry or exit point using the FlexConnect notation.

### WCP-22 Recursion

A task has the ability to invoke itself. The parent task cannot complete until all child tasks are complete. There must be at least one path that is not self-referencing and terminates normally to ensure that the task does not invoke itself infinitely.

In FlexConnect, an object has the ability to create an instance of itself. However, it is not possible to specify an end condition that ensures that an object does not invoke itself an infinite number of times. Thus, this pattern is partially supported by FlexConnect.

### WCP-23 Transient Trigger

If the activity to which the trigger is pointing is not waiting for the trigger at the time that the trigger is received, then the trigger is lost. Trigger must be acted upon immediately. Trigger can be safe (one instance waiting for a trigger) or not safe (multiple instances).

Not supported by FlexConnect.

### WCP-24 Persistent Trigger

Relevant parts of a sub process are completed and the sub process continues without blocking the "mother" process. The trigger can be either buffered or can initiate a task directly.

Signals are used for synchronisation in situations between parent and child state machines that run asynchronously. If the children reach the end of their lifecycle then they will send a finish signal. At this point in time the parent state machine might not be in a state to wait for the children, so the signals are persisted to allow them to can be received at the stage when the parent is ready to receive them. Thus, FlexConnect fully supports this pattern.

### WCP-25 Cancel Region (disable an activity set)

Disable a set of activities within an instance. Any executing activities are stopped. Activities can be part of any process.

Not supported by FlexConnect.

### WCP-26 Cancel Multiple Instance Activity

Cancel any multiple instance tasks that have not been completed and move to the subsequent task.

Partially supported by FlexConnect. Individual instances of a multiple instance activity can be cancelled but there is no facility to instantly cancel all active instances of a particular multiple instance activity.

### WCP-27 Complete Multiple Instance Activity

Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronise the instances at completion before any subsequent activities can be triggered.

Not supported by FlexConnect.

### WCP-28 Blocking Discriminator

Two or more branches converge into a single branch following an earlier divergence. Control is passed on once the first active incoming branch is enabled for the same process instance. Discriminator resets when all active incoming branches are enabled for the same process instance. Subsequent enabling of incoming branches is blocked until the discriminator has reset.

Not supported by FlexConnect.

### WCP-29 Cancelling Discriminator

Triggering the discriminator cancels the execution of incoming branches and resets the construct. Issue exists with determining the boundaries of the cancellation region.

Not supported by FlexConnect.

### WCP-30 Structured Partial Join

Merge two or more branches into a single branch, once 'N' incoming branches have been enabled. Subsequent active incoming branches do not result in flow of control and are ignored. Join construct resets when all active incoming branches have been enabled.

This pattern is supported by FlexConnect. Synchronisation occurs at a structured partial join when a "threshold" number of state machine instances have completed. The lifecycle of the state machine then continues after "N-out-of-M" children have finished and subsequent children are ignored.

### WCP-31 Blocking Partial Join

The convergence of two or more branches into a single branch. The thread of control is passed after 'N' of the incoming branches has been enabled. The join construct resets when all active incoming branches have been enabled for the same process instance. Subsequent enablements of incoming branches are blocked until the join has reset.

Not supported by FlexConnect.

### WCP-32 Cancelling Partial Join

Two or more branches converge into a single branch. The thread of control is passed after 'N' of the incoming branches has been enabled. Triggering the join cancels the execution of all other incoming branches and resets the construct.

Not supported by FlexConnect.

### WCP-33 Generalised AND-Join

Converge two or more branches into a single branch and pass thread of control on when all input branches are enabled. Additional triggers received on one or more branches are persistent and are retained for future firings.

Not supported by FlexConnect.

### WCP-34 Static Partial Join for Multiple Instances

Multiple concurrent instances of an activity are created. The required number of instances is known when the first activity commences. Once 'n' instances have completed, the next task is triggered and subsequent completions of any remaining instances have no effect.

Thus pattern is supported by FlexConnect by setting the “threshold” property on a signal to a number greater than zero (n), which synchronises when n-out-of-m instances of a state machine have completed. The remaining active 'n' instances complete their lifecycle with no effect on the process.

### WCP-35 Static Partial Join for Multiple Instances with Cancellation

Multiple concurrent instances of an activity are created and the required number of instances is known when the first activity starts. Once 'n' activity instances have been completed, trigger the next task and cancel all remaining instances.

Not supported by FlexConnect, since multiple instances of tasks or state machines cannot be cancelled as a group.

### WCP-36 Dynamic Partial Join for Multiple Instances

Multiple concurrent instances of an activity are created and the required number of instances is determined by some external factor such as state data, resource availability or process communications and is not known until the final instance is complete. It is possible to initiate additional instances dynamically. Once 'n' activities are complete, the next task is triggered. Any other active instances can be completed but will be ignored.

Partially supported. In FlexConnect, child instances can be triggered in parallel without a priori knowledge because child objects have a relationship with a parent object. The relationships between a parent and its children can also be interrogated to determine if 'n' child instances have finished since there is a threshold property on each signal involved in the relationship. The remaining active 'n' instances are allowed to complete their lifecycle. This applies generally in FlowConnect to support the "partial join" patterns. This pattern is partially supported by FlexConnect because the lower-and upper-bounds for multiple instances of state machines is specified using thresholds instead of expressions.

### WCP-37 Acyclic Synchronising Merge

The convergence of two or more branches into a single branch that passes the thread of control onto the subsequent branch when each incoming branch is enabled. Where a given branch does not have a thread of control passed to it at the divergence, "false tokens" are passed along the branch to ensure that the merge construct can determine which incoming branches are to be synchronised.

Not supported by FlexConnect because "false tokens" are not part of the FlexConnect language.

### WCP-38 General Synchronising Merge

The convergence of two or more branches where the thread of control is passed to the subsequent branch when each active incoming branch has been enabled or it is not possible that the branch will be enabled at any time in the future.

This pattern is not supported by FlexConnect since it is only possible to specify structured process models in FlexConnect, meaning that there is no need to support this pattern.

### WCP-39 Critical Section

Two or more connected sub-graphs are identified as "critical sections". For any given process instance, only activities in these critical sections can be active at a given time. Once execution of the activities in one critical section commences, it must complete before another critical section can commence.

This pattern is not supported by FlexConnect.

### WCP-40 Interleaved Routing

Each member of a set of activities must be executed only once. They can be executed in any order but no two activities can be executed at the same time. Once all of the activities have been completed, the next activity can be commenced.

Not supported by FlexConnect, due to the context condition “no two activities can be executed at the same time”.

### WCP-41 Thread Merge

At a given point in a process, a nominated number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution.

In FlexConnect the Thread Merge pattern is supported by the *message* and *finish* signal types, which are used synchronise instances of an state machine at a gateway.

### WCP-42 Thread Split

At a given point in a process, a nominated number of execution threads can be initiated in a single branch of the same process instance.

Supported by the *spawn* signal type in FlexConnect, which initiates a number of instances of a particular state machine according to the upper-bound of the signal. In this context each instance is viewed as a thread.

**WCP-43 Explicit Termination**

A given process (or sub-process) instance should terminate when it reaches a nominated state. Typically this is denoted by a specific end node.

Supported by FlexConnect since this language has a specific end node, which is a final state.

### 6.1.8 Control-flow Patterns Evaluation Summary

This control-flow pattern evaluation of FlexConnect has demonstrated how business objects and state machines can be used to model processes using the modelling syntax that has been introduced and explained in this thesis. The local semantics of the state-based approach in FlexConnect allows the language to present solutions to pertinent issues that are often identified by control-flow pattern evaluations, such as the creation and synchronisation of multiple instances and synchronisation at an OR-Join using a parent/child relationship. The results of the control-flow pattern evaluation of FlexConnect is summarised in Table 6.1 and Table 6.2. A good level of pattern support is seen in the first 20 patterns (17/20 patterns supported), which correspond to the first set of 20 workflow patterns from [6]. A decreased level of pattern support is evident in the second set of 23 extended workflow patterns (6 patterns fully supported, 3 patterns partially supported), which capture more specific features of a workflow system.

A FlexConnect process model is a routing pattern for a process that defines the states in a process and the transitions that may be performed between states in a process. At a higher level in FlexConnect, state machines are grouped into business objects. Business objects are entities in FlexConnect that play a role in a system and will be involved in one or more business processes. To capture the workflow patterns using the object-centric approach of FlexConnect, heavy use is made of signals instead of transitions to connect nodes in an object-centric model together. It can be seen from this pattern-based evaluation that usage of signals (connecting gateways) as

| ID | Pattern Name | FlexConnect |
|---|---|---|
| WCP-1 | Sequence | + |
| WCP-2 | Parallel Split | + |
| WCP-3 | Synchronisation | + |
| WCP-4 | Exclusive Choice | + |
| WCP-5 | Simple Merge | + |
| WCP-6 | Multi-choice | + |
| WCP-7 | Structured Synchronising Merge | + |
| WCP-8 | Multi-Merge | + |
| WCP-9 | Structured Discriminator | + |
| WCP-10 | Arbitrary Cycles | + |
| WCP-11 | Implicit Termination | - |
| WCP-12 | MI without Synchronisation | + |
| WCP-13 | MI with a priori Design Knowledge | + |
| WCP-14 | MI with a priori Runtime Knowledge | + |
| WCP-15 | MI without a priori Runtime Knowledge | + |
| WCP-16 | Deferred Choice | + |
| WCP-17 | Interleaved Parallel Routing | - |
| WCP-18 | Milestone | + |
| WCP-19 | Cancel Activity | + |
| WCP-20 | Cancel Case | - |

Table 6.1: Summary of Control-flow Pattern Support (Patterns 1-20)

opposed to transitions (connecting states) is what enables many control-flow patterns to be supported by the object-centric approach.

The syntax of FlexConnect is very different to the usual syntax of activity-centric modelling approaches. FlexConnect does not have an explicit modelling notation for the set of control-flow splits and joins that are usually found in activity-centric process modelling languages such as YAWL (i.e. AND-Split, AND-Join, XOR-Split, etc). The various types of splits and joins are supported by sending and receiving signals at gateways using the mode and configuration gateway properties to specify control-flow behaviour, often for the purpose of creating new objects. This represents an additional burden to a process designer, who will have to compose a process primarily in terms of the objects that the process is constructed of, rather than in

| ID | Pattern Name | FlexConnect |
|---|---|---|
| WCP-21 | Structured Loop | - |
| WCP-22 | Recursion | +/- |
| WCP-23 | Transient Trigger | - |
| WCP-24 | Persistent Trigger | + |
| WCP-25 | Cancel Region | - |
| WCP-26 | Cancel MI Activity | +/- |
| WCP-27 | Complete MI Activity | - |
| WCP-28 | Blocking Discriminator | - |
| WCP-29 | Cancelling Discriminator | - |
| WCP-30 | Structured Partial Join | + |
| WCP-31 | Blocking Partial Join | - |
| WCP-32 | Cancelling Partial Join | - |
| WCP-33 | Generalised AND Join | - |
| WCP-34 | Static Partial Join for MI | + |
| WCP-35 | Cancelling Static Partial Join for MI | - |
| WCP-36 | Dynamic Partial Join for MI | +/- |
| WCP-37 | Acyclic Synchronising Merge | - |
| WCP-38 | General Synchronising Merge | - |
| WCP-39 | Critical Section | - |
| WCP-40 | Interleaved Routing | - |
| WCP-41 | Thread Merge | + |
| WCP-42 | Thread Split | + |
| WCP-43 | Explicit Termination | + |

Table 6.2: Summary of Control-flow Pattern Support (Patterns 21-43)

terms of the sequencing of tasks. It can be recognised that this approach to process modelling might not be desirable in all situations.

However, some benefits are witnessed from using an object-centric approach. This approach presents a simple way to capture the OR-Join, which is often a difficult pattern to implement in workflow languages because it presents a difficult synchronisation problem where the number of active incoming arcs to the join must be known. The OR-Join should block control-flow until all active incoming arcs have either arrived or have been cancelled. Using an object-centric approach to the OR-Join, a parent waits until it has received all signals from all active children. Support for the OR-Join is achieved at a gateway simply because a parent knows the number and type of all active children it expects to receive signals from. This approach eliminates the need to trace back through a model to discover if an incoming branch can still complete, because a relationship stores this information.

## 6.2 Taxonomy of Flexibility Evaluation

In this section we evaluate the FlexConnect and YAWL languages (including the YAWL Worklets service) using the patterns in the Taxonomy of Flexibility [90]. The categories included in the Taxonomy of Flexibility are Flexibility by Design, Flexibility by Deviation, Flexibility by Underspecification and Flexibility by Change. FlexConnect and YAWL have been evaluated in order to compare and contrast how flexibility can be achieved by representatives of both object-centric and activity-centric process modelling languages.

### 6.2.1 Flexibility by Design

Flexibility by Design is the ability to select the most appropriate execution path from a set of design-time alternatives to be made at runtime within a process model. The Flexibility by Design patterns are Parallelism, Choice, Iteration, Interleaving, Multiple Instances and Cancellation.

#### Parallelism

FlexConnect supports the Parallelism pattern in the same way it supports the AND-Split control-flow pattern, as shown in Figure 6.2. An instance of a child state machine is created by a “spawn signal”. A spawn signal is sent from either a pre- or post-gateway of a state to the initial state of another state machine. This gives FlexConnect the ability to execute child several tasks in parallel, as well as in parallel to the parent lifecycle. YAWL supports parallelism with a dedicated AND-split operator that enables two or more tasks following the completion of a task, as shown in Figure 6.18.

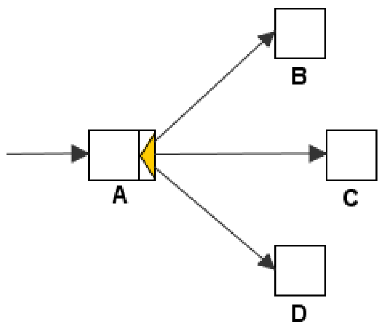


Figure 6.18: Parallelism in YAWL

### Choice

The Choice pattern is supported in the same way as the OR-Split control-flow pattern is supported. This is, the ability to choose between zero-to-many state machines from a number of different types of state machines allows the choice at runtime to be made as to which combination of state machines will be created, as shown in Figure 6.7. In the example provided, four choices can occur as the outcome of the split: 1) An instance of SM02 is created, 2) An instance of SM03 is created, 3) An instance of SM02 and SM03 is created, and 4) No instances of either SM02 or SM03 are created. YAWL supports the choice pattern with an XOR-split (or OR-split) that allows a choice of task (B, C or D) to be made following the completion of a task (A), as shown in Figure 6.19.

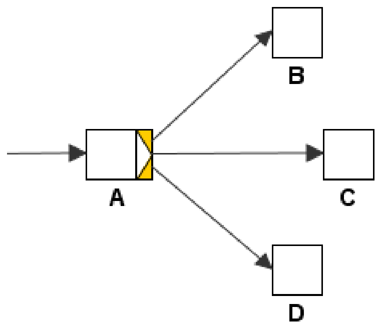


Figure 6.19: Choice in YAWL

### Iteration

The Iteration pattern is supported by a state with an outgoing transition that has the same state as its target. This means that, depending on the evaluation of conditions at runtime, the path of execution may iterate over the task contained in State 001, as shown in Figure 6.20. This pattern is supported in the same manner by YAWL because control-flow in YAWL can iterate over a task or a set of tasks.

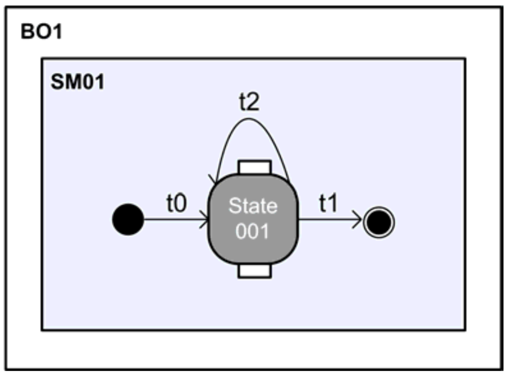


Figure 6.20: Iteration in FlexConnect

### Interleaving

Interleaving is not supported because it is not possible to select one-amongst-many active children to execute at a particular moment in time. YAWL supports parallel interleaving with the use of a condition that behaves as a semaphore. As shown in Figure 6.21, this allows only one instance of a task among a set of tasks (e.g. tasks A, B, C and D) that share the semaphore to execute at one time.

### Multiple Instances

Since FlexConnect supports the *Multiple Instances without a priori runtime knowledge* control-flow pattern (WCP-15), all multiple instance pattern variants are supported since they can be seen as variations/simplifications of

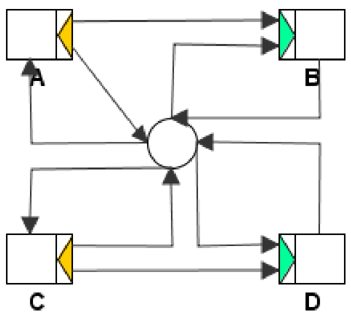


Figure 6.21: Parallel Interleaving in YAWL

WCP-15. The solution for the WCP-15 multiple instance pattern is shown in Figure 6.15. At runtime, a number of instances are created as needed due the unbounded ($n$) upper bound of the spawn signal. This allows any number of instances to be created which may be dependent on state data or resource availability at runtime. Synchronisation of all created instances of SM02 is performed at the pre-gateway of State 004, which *waits-for-all* created instances of SM02 to complete. Also, if there was no finish signal Sig02 to indicate the completion of SM02 lifecycles, then this provides support for multiple instance creation with no synchronisation. Support for the multiple instances pattern in YAWL is shown Figure 6.22. After task A has been completed, the expressions xq1 and xq2 are evaluated at runtime to determine the lower- and upper-bound number of instances of task B that should be created.

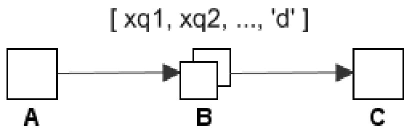


Figure 6.22: Multiple Instances in YAWL

#### Cancellation

FlexConnect supports cancellation of task instances at runtime. A cancelled task instance can then be restarted. Cancellation of a state machine instance has wider ramifications since if the state machine is a parent of any child state machines instances, the child state machine instances are also cancelled. This is not supported. In YAWL, the execution of a task can result in the cancellation of another task, tasks or a case. As shown in Figure 6.23, the execution of task B causes task A to be cancelled.

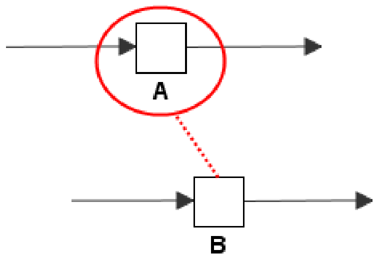


Figure 6.23: Cancellation in YAWL

### 6.2.2 Flexibility by Deviation

Flexibility by deviation allows control-flow in a process model to deviate from the prescribed path of execution. The advantage of allowing deviations is to permit a process to accommodate situations at runtime that require an alternate sequence of tasks to be performed. The Flexibility by Deviation patterns are Undo, Redo, Skip, Create Additional Instance and Invoke Task.

#### Undo

Not supported by FlexConnect or YAWL. However, the description of the Undo pattern "One point to consider with this operation is that it does not imply that the actions of the task are undone or reversed" taken from Schonenberg *et al*, leads one to believe that Undo can be supported by cancelling a task, then restarting it. We consider that this quote is not really true of an Undo operation (since nothing is 'undone'), so this pattern is not supported.

### Redo

Not supported by FlexConnect or YAWL. In the FlowConnect engine (from where FlexConnect is inspired), there is a "hammer" concept that allows a user with appropriate privileges to move the state of execution of an object to any previous state. However, this is more of an administration feature than a language feature.

### Skip

Partially supported by FlexConnect but not supported by YAWL. If a task is marked as an 'optional' task, then its completion is not required for a state to exit. Hence, an optional task can be 'skipped' by simply stopping work on the task. The task will persist until it is either completed at a later stage or it is cancelled. It is not possible to skip a 'compulsory' task. There is no explicit Skip operation in FlexConnect, however partial support for this pattern is claimed due to optional tasks that need not be completed.

### Create Additional Instance

Supported by FlexConnect but not by YAWL. Creating additional instances of a state machine (that may represent a single task or many tasks), is done by defining a creation region and linking a creation signal from the region to a JOB (Job object). At any time when the state machine instance is 'in' the region, additional instances of the JOB can be created. A region can be of any size, meaning it can contain anywhere from a single state to all states in a state machine. The number of instances created may be restricted by the lower/upper bounds of the creation signal. As shown in Figure 6.24, from *1 to n* Client Interaction instances, *0 to n* Child Support instances and *0 to 1* Rental Assistance instances can be created when the state machine lifecycle of the Client Intake COROB (Coordination Object) is in the Case Management Region.

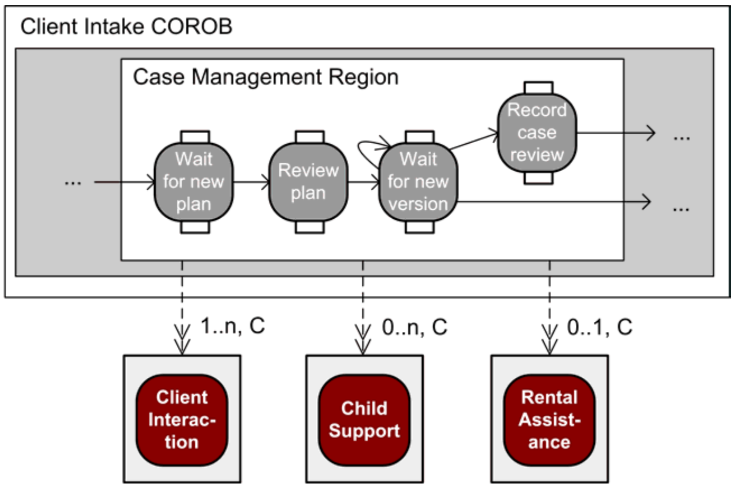


Figure 6.24: Create Additional Instance in FlexConnect

### Invoke Task

The Invoke Task pattern is not supported by either FlexConnect or YAWL. It is not possible to invoke/execute tasks in a FlexConnect process model on an adhoc basis. Task invocation can only be done after a transition has been performed to the state that contains the task. Since a state machine can only be in one state at one time, support for the Invoke Task pattern could potentially require to exit the current state and take a transition to *any* state, regardless of whether there is a transition connecting the source and target states. Undesirable side-effects on control-flow synchronisation (in particular) could be experienced as a result. The same limitation is witnessed in YAWL Worklets.

## 6.2.3 Flexibility by Underspecification

Flexibility by Underspecification is the ability to execute and complete a process model that contains insufficient design-time information. The Flexibility by Underspecification patterns are Late Binding, Late Modelling, Static – before placeholder execution, Dynamic – before placeholder execution, Static – at placeholder execution and Dynamic – at placeholder execution.

### Late Binding

From Schonenberg *et al*, a summary of Flexibility by Underspecification is given as "This approach to process design and enactment is particularly useful where distinct parts of an overall process are designed and controlled by different work groups, but the overall structure of the process is fixed". In FlexConnect, there is ability to send work that is outside the scope of a given state machine to an alternative state machine that can handle the out-of-scope work in question. This ability is known as a *referral.* From a creation region in a state machine, a referral signal can be sent to a ROB (Referral Object), from where an instance of a state machine (possibly from a set of state machines) can be created. For example, in Figure 6.25, an instance of a ROB can create a nested instance of a Gambling Issue COROB, Alcoholism COROB or Legal Support COROB, depending on whether any of the issues that these COROBs deal with appeared during the Case Management Region of the Client Intake COROB. Essentially, this is a late binding mechanism that allows the target of a referral to remain loosely defined at design time.

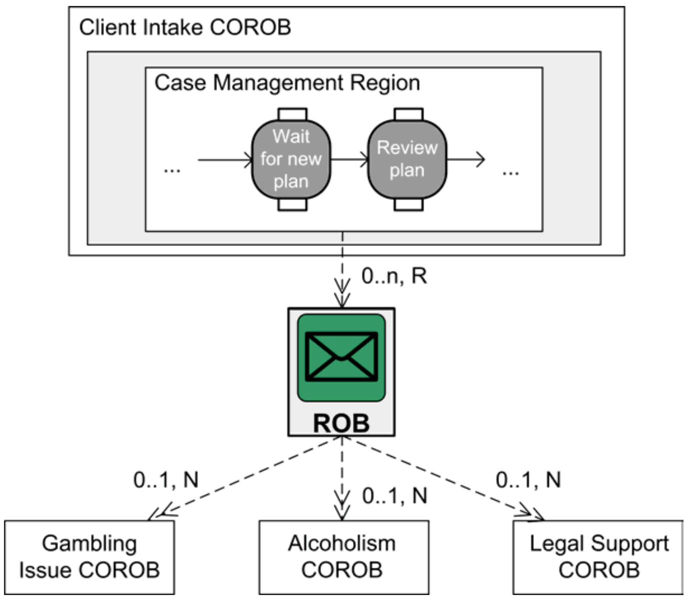


Figure 6.25: Late Binding in FlexConnect

YAWL Worklets also support late binding since a worklet-enabled task can act as a task 'placeholder', which enables the type of the task to be executed to be dynamically selected at runtime. This ability permits YAWL Worklets to support late binding without structurally altering the model.

### Late Modelling

Not supported by FlexConnect, since process fragments cannot be modelled at runtime. YAWL Worklets support late modelling of process fragments by allowing a process model to be created at runtime in response to a situation where a suitable worklet cannot be found. Parts of a process model can be left undefined as task 'placeholders' which are to be modelled at runtime. This allows process modelling to be deferred until the last possible moment at runtime.

#### Static – before placeholder execution

In FlexConnect, the content and boundaries of a creation region is defined at design time. The content or boundaries of a creation region cannot be changed at run time. A static placeholder, defined before placeholder execution is a creation region that consists of a single state. This restricts the creation region to be enacted at the same process fragment for every instance of the process. This pattern is not supported by YAWL.

#### Dynamic – before placeholder execution

A dynamic placeholder, defined before placeholder execution is a creation region that consists of more than one state. This has the effect of granting a state machine the ability to send signals from the creation region while the instance is in any state of the process that belongs to the region for every instance of the process. This pattern is not supported by YAWL.

#### Static – at placeholder execution

Not supported by FlexConnect or YAWL.

#### Dynamic – at placeholder execution

Not supported by FlexConnect. However, YAWL Worklets support this pattern by specifying a worklet-enabled task at design-time and then allowing the type of task to be realised at the placeholder to be dynamically chosen again each time that the placeholder is executed.

### 6.2.4 Flexibility by Change

Flexibility by change is the ability to modify a process model at runtime such that one or all of the currently executing process instances are migrated to a new process model. The Flexibility by Change patterns are Momentary

Change, Evolutionary Change, Entry Time, On-the-fly, Forward Recovery, Backward Recovery, Proceed and Transfer.

### Momentary Change

Momentary change is supported by FlexConnect with the use of creation regions. A creation region enables several types of momentary change to occur, where a momentary change is a change that affects one or more selected process instances (i.e. the occurrence of an isolated incident). Momentary change can include the creation of an *unplanned* task, which is a type of momentary change that can be modelled and enacted by FlexConnect using a *creation signal*. YAWL does not support momentary change.

### Evolutionary Change

Evolutionary change is not supported by FlexConnect. YAWL Worklets support evolutionary changes to a process model with the use of ripple-down rules. Ripple-down rules can be specified at runtime and are used to organise worklets into a hierarchy of tasks which are structured according to the condition(s) that must be satisfied for a particular worklet to be enabled. Over time, new ripple-down rules can be added and old rules can be altered, allowing evolutionary change to affect a YAWL process model.

### Entry Time

Entry Time is partially supported in FlexConnect by including the initial state of a state machine as the only state in a creation region, which is also the only creation region for that state machine. This is the only way in a FlexConnect model to localise change in a state machine to entry time. This pattern is not supported by YAWL.

### On-the-fly

On-the-fly change is supported by FlexConnect. Enactment of a dynamic signal from a creation region can occur at any time when a state machine instance is in the creation region. This enables change to be handled as it occurs in a region, which represents a reactive approach to flexibility. YAWL Worklets support on-the-fly change because a process can be customised at any time during its execution and new processes are executed according to any customisations that have been made during previous executions.

### Forward Recovery

Forward recovery (process instances affected by a change are aborted) is not supported by FlexConnect. Running processes that are affected by change can be aborted in YAWL Worklets.

### Backward Recovery

Backward recovery (process instances affected by a change are aborted, then restarted) is not supported by FlexConnect. Running processes that are affected by change can be aborted and then restarted in YAWL Worklets.

### Proceed

FlexConnect does not maintain a particular 'way' of handling change that is updated over time. Instead, all possible options are presented to a user at runtime, and the appropriate type of change needed to fit a situation confronting a user is chosen at the moment it is needed. No distinction is made in FlexConnect between *existing* instances that are handled the *old* way and *new* instances of the same process that are handled the *new* way. YAWL also does not support the Proceed pattern.

**Transfer**

The transfer pattern is supported in FlexConnect by *delegation*, which is the ability of a user to trigger the transfer of context and data from an executing task to a different task. Delegation provides support for circumstances that may change over time (i.e. if a problem appears during a client interaction, delegate the interaction to a task that can support the problem). To support such situations, a new (*delegatee*) takes over execution of a previous (*delegator*), which corresponds to the *transfer* of an existing process instance to a new instance. Delegation can only be performed while a state machine is in a creation region that sends a *delegation signal.* Support for the transfer pattern in YAWL Worklets is much more straight-forward, since this pattern supported by allowing a process fragment to replace another fragment which allows a task instance to be transferred to another task instance.

### 6.2.5 Flexibility Patterns Evaluation Summary

In this section we have illustrated how FlexConnect and YAWL support patterns in the Taxonomy of Flexibility. We note that support for the Flexibility by Design patterns is achieved in the same way as particular control-flow patterns for both languages. The other categories (Flexibility by Deviation, Underspecification and Change) allow us to present the usage of other aspects of YAWL (through the YAWL Worklet service) and FlexConnect (by using dynamic signals and creation regions).

As we can see in Table 6.3, both FlexConnect and YAWL achieve support for a number of flexibility patterns. For both languages, support for *all* patterns for any particular category of flexibility in the taxonomy is not achieved. However, FlexConnect achieves support for a *spectrum* of flexibility patterns, since we can observe that FlexConnect supports a number of patterns in *each* category. This range of support for a variety of patterns indicates that the FlexConnect modelling language can be applied in a range of situations and allows us to highlight the versatility of the language.

| Category | Pattern | FlexConnect | YAWL |
|---|---|---|---|
| **Flexibility by Design** | | | |
| | Parallelism | + | + |
| | Choice | + | + |
| | Iteration | + | + |
| | Interleaving | - | + |
| | Multiple Instances | + | + |
| | Cancellation | + | + |
| **Flexibility by Deviation** | | | |
| | Undo | - | - |
| | Redo | - | - |
| | Skip | +/- | - |
| | Additional Instance | + | - |
| | Invoke Task | - | - |
| **Flexibility by Underspecification** | | | |
| | Late binding | + | + |
| | Late modelling | - | + |
| | Static – before PH | + | - |
| | Dynamic – before PH | + | - |
| | Static – at PH | - | - |
| | Dynamic – at PH | - | + |
| **Flexibility by Change** | | | |
| | Momentary Change | + | - |
| | Evolutionary Change | - | + |
| | Entry Time | +/- | - |
| | On-the-fly | + | + |
| | Forward Recovery | - | + |
| | Backward Recovery | - | + |
| | Proceed | - | - |
| | Transfer | + | + |

Table 6.3: Summary of Taxonomy of Flexibility Support

Support for the Flexibility by Design patterns that are presented in Schonenberg *et al* [89] is achieved by FlexConnect and YAWL in the same way that related control-flow patterns are supported. This is because of the close similarities that can be observed between the Flexibility by Design patterns to particular control-flow patterns. Thus, support for all Flexibility by Design patterns except the Interleaving pattern is achieved in FlexConnect. The Interleaving pattern is not supported for the same reason that the "Parallel Interleaved Routing" control-flow pattern is not supported. YAWL supports all of the Flexibility by Design patterns.

To support the Flexibility by Underspecification patterns, a process modelling language must have a special type of node known as a *placeholder*. A placeholder is essentially a black box that represents an incomplete or underspecified fragment of a process model. The closest concept in FlexConnect to a placeholder (as it is described in Schonenberg *et al*) is a *creation region*. The main difference between a creation region and a placeholder is that a placeholder is an underspecified node that has unknown content. In contrast, the content of a creation region is well-defined (it contains one or more states), however the purpose of a creation region is to enable *unplanned* task creation. Thus, the rationale behind a creation region is the same as a placeholder, since a creation region identifies the fragments of a process model that are underspecified and allows runtime adjustments to be made to the process at the nodes that are contained within the region. A YAWL Worklet can be seen as a type of placeholder, which allows YAWL to support many of the Flexibility by Underspecification patterns.

The evaluation shows that flexibility in either language can only be achieved by incorporating extensions to the core of either language. For example, to support aspects of Flexibility by Deviation, Flexibility by Underspecification and Flexibility by Change, dynamic signals and creation regions are used in FlexConnect and the Worklet service is used in YAWL. Using the Taxonomy of Flexibility as a tool for comparison with other process modelling languages, we can observe that FlexConnect compares well with YAWL in general, especially in the Flexibility by Deviation category. On the basis of this evaluation and comparison with YAWL we conclude that FlexConnect is a *general-purpose* flexible process modelling language.

# Chapter 7

# Epilogue

In this thesis an alternative approach for the design of process models was proposed, in which models are modularised along key business objects rather than along activity decompositions. This epilogue summarises the contributions of the thesis, identifies how the criteria for a solution were realised in the thesis and concludes with a discussion of future work.

## 7.1 Summary of Contributions

In this thesis a meta-model for object-centric (O-C) process modelling was introduced, analysed and discussed. To empirically test the meta-model, an O-C modelling tool was developed called *FlexConnect*. The modelling tool was used to capture two large industrial scenarios and the process modelling capabilities of the approach were evaluated using the control-flow patterns and the taxonomy of flexibility. A FlexConnect model focuses upon identifying the objects (or artifacts) that are involved in a process and specifies the relations between them. Object lifecycles are decomposed following an object-oriented paradigm, with higher-level object lifecycles spawning instances of child objects. The focus on decomposition gives three main benefits to the approach, which have been introduced and discussed over the course of this thesis.

Firstly, an O-C approach is a way of modelling that promotes *modularisation* of a process model into objects. An object is an entity in a process that captures related tasks which share some association such as common data or purpose. For example, the tasks required to complete an Inspection are captured by an Inspection object. The modularisation approach promoted by activity-centric approaches is restricted in terms of the types of decompositions it allows. The units of decomposition in this approach are basically functional black-boxes: at runtime, after an activity (task or sub-process) is started the process waits for the activity to complete. When modularising in terms of an O-C approach, an object instance has a lifecycle that manages interaction with other objects at different points in the lifecycle in addition to task execution. This latter approach to modularisation is richer. An O-C approach enables modellers to modularise along the key objects (or entities) of a domain. This enables changes in the requirements to be addressed within the specification of these entities. In contrast, in an activity-centric decomposition approach, any changes affecting one entity might also affect many sub-processes.

Secondly, precise control-flow dependencies *between* business objects (modules) are easily identified. In a FlexConnect model these dependencies are captured using gateways and signals. If an object creates instances of other objects, a parent/child relationship is established from which the objects that *belong* to each other can be derived. This has the advantage of enabling a parent object to know the number and type of children it should wait for at a point of synchronisation (a gateway). Synchronisation behaviour can be further refined by setting gateway and signal properties (by specifying the gateway configuration, gateway mode and signal mode).

Lastly, the FlexConnect modelling language is a *flexible* process modelling language. A flexible process model is one that has the ability to cope with several types of change. We demonstrated how FlexConnect handles several types of flexibility such as *unplanned* task creation, task delegation and task referral by defining an object-oriented framework. As illustrated in Figure 7.1, a strength of FlexConnect is that a *spectrum* of flexibility patterns are supported across the taxonomy of flexibility (defined in [89]). This indicates that the FlexConnect modelling language can be applied in a range of situations and highlights the versatility of the language. Using the taxonomy of flexibility as a tool for comparison with other process modelling languages, FlexConnect compares well with Declare and YAWL Worklets, and compares favourably with ADEPT1 and FLOWer, especially in the category "Flexibility by Design".

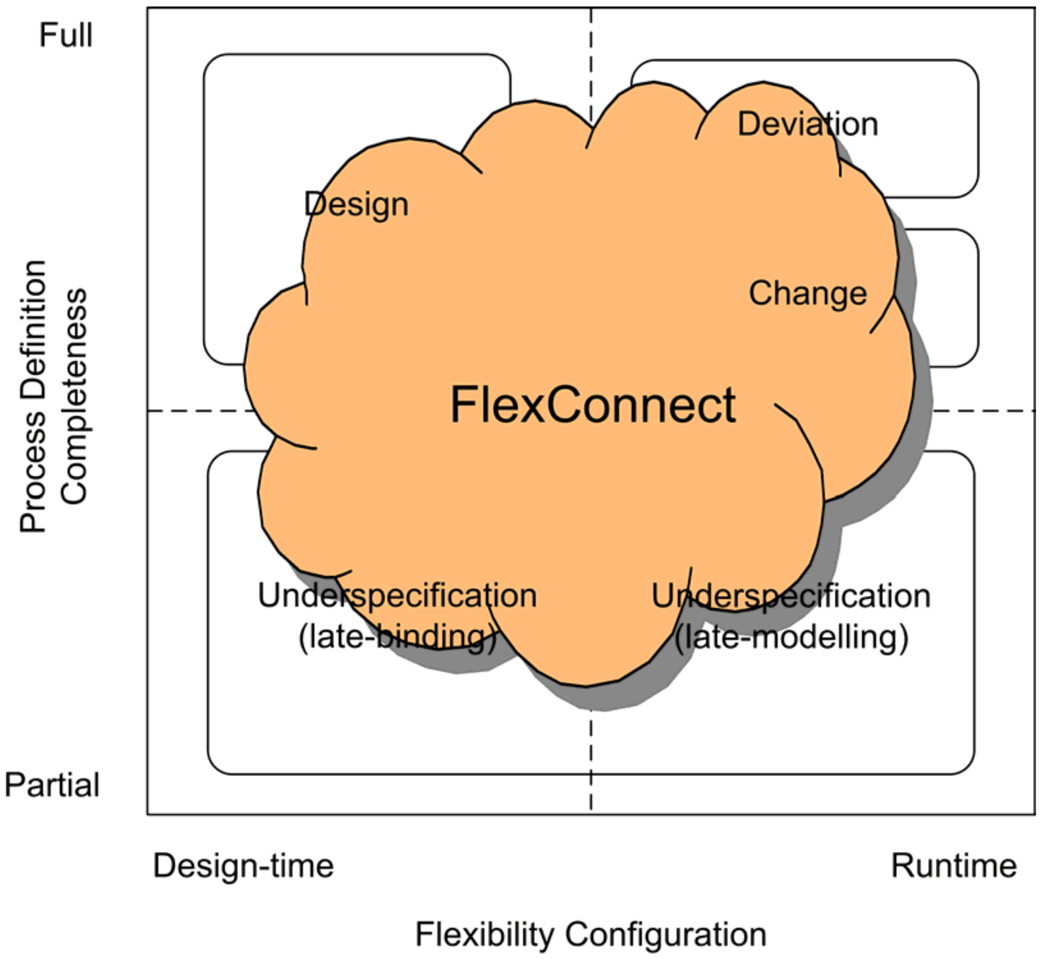

Full
Partial
Process Definition Completeness
Design
Deviation
Change
FlexConnect
Underspecification (late-binding)
Underspecification (late-modelling)
Design-time
Runtime
Flexibility Configuration

Figure 7.1: Flexibility Patterns Coverage of FlexConnect

## 7.2 Fulfillment of Solution Criteria

In Chapter 1, six *criteria for a solution* were introduced. Below we recall the criteria and identify how each criterion was realised in the thesis.

**1. The language must be defined in terms of a meta-model.** *In this thesis we aim to define an object-centric language to process modelling and investigate applications of the language. The elements of the approach and their associations should be captured as a meta-model to provide the necessary syntax for object-centric process modelling.*

A meta-model was developed to provide a foundation for the development of O-C process models. In addition, an object-to-activity conversion procedure was defined to transform control-flow from an O-C to an activity-centric representation, in this case, to produce a YAWL model from an O-C model. The motivation for providing such a transformation is to show the relations between O-C and activity-centric process modelling approaches from the control-flow perspective. In particular, the transformation shows that it becomes difficult or impossible to maintain the process model structure (i.e. its decomposition into objects) when objects communicate in an arbitrary manner. Specifically, the decomposition is maintained if the objects communicate only through spawn and finish signals.

**2. The language must be flexible.** *The modelling language must have the ability to capture scenarios that experience ad-hoc change and case-by-case variation.*

The FlexConnect language allows for the flexible instantiation of artifacts in a process model as they are required. This language reflects the different needs that can confront different cases of the same process during runtime. We demonstrated how the FlexConnect meta-model supports the design of processes consisting largely of *unplanned* activities. In particular we showed how three fundamental business object types (Coordination Object, Job Object and Referral Object) can be combined to capture different flexibility requirements. The key underlying principle is that a FlexConnect model specifies "what can happen during a case", rather than "how it should happen". Any constraints regarding which objects can be created and when, are overlaid on top of the object model. This approach contrasts with mainstream process modelling paradigms based on flowchart-like notations, in which the activities to be performed and their control-flow relations form the backbone of a process model. Of course, while flexibility is essential in domains such as human services there are situations where this flexibility should be constrained. Thus, the approach also supports the definition of 'thresholds' to constrain the number of children that a parent can create.

**3. The language must have a graphical modelling notation.** *The language must be visual, based on a syntax that is captured as a meta-model that should aid the design of object-centric process models.*

A graphical notation was developed on the basis of the FlexConnect process modelling syntax defined in the meta-model (see Criterion #1). The notation was introduced in two parts in this thesis. Firstly, the elements of the notation to model structured processes models were introduced in Chapter 3. A minimalist set of elements to capture flexible process models were introduced in Chapter 4. The structured and flexible aspects of the notation were applied to capture industrial O-C process models provided by the project's industrial partner.

**4. The language must be embodied in a modelling tool.** *The language must be embodied in a modelling tool to provide a computer-assisted way of constructing and validating flexible object-centric process models.*

A modelling tool called FlexConnect was developed to embody the meta-model and notation. FlexConnect allows a process modeller to capture both structured and flexible process models using the graphical notation that was proposed for O-C process models. The tool has the functionality to transform control-flow defined in a FlexConnect model to a YAWL model based on the object-to-activity model conversion procedure, which was implemented using customised plugins from the ProM framework [24]. The tool also has the functionality to generate an initial marking for a Coloured Petri Net (CPN) that acts as an interpreter for the execution of FlexConnect process models.

**5. The language must have a formal grounding.** *A formal foundation serves the purpose of removing ambiguity regarding the runtime behaviour of all the modelling elements of the language. On the basis of the formal foundation, the ability to reason about the language and test process models developed using the language to validate behavioural correctness should be possible.*

The semantics of the approach is formally captured by a CPN. The CPN provides several insights into how the semantics of a flexible modelling approach can be formally defined, which has the advantage of achieving flexibility without adding overly complex business rules to maintain control over process behaviour. Besides providing a grounding of the proposal, the CPN may be used for at least two purposes: **1) Behavioural Correctness.** The CPN enables one to examine a FlexConnect model for behavioural correctness by executing the model in CPN Tools. **2) Simulation.** Simulation of a FlexConnect model can be performed using CPN Tools to validate the expected runtime behaviour of the model.

**6. The language must be 'suitable'.** *The language must have the ability to support recurring patterns of control-flow in process models and have the ability to support a range of patterns of flexibility.*

An evaluation of the FlexConnect language in terms of the control-flow patterns [6] and taxonomy of flexibility [89] is provided, which highlights the strengths and weaknesses of FlexConnect as a representative O-C process modelling language. The evaluation demonstrated the ability of FlexConnect to capture advanced synchronization patterns such as the OR-Join (in certain contexts) and multi-instance activity patterns. Several weaknesses were also demonstrated such as the requirement that AND-Splits can only be captured by decomposing a business object into sub-objects. This is an approach that might not be desirable in some settings because the comprehensibility of large models may be affected if each branch from an AND-Split requires the creation of a new object. An evaluation of FlexConnect using the taxonomy of flexibility shows that types of flexibility are supported in all categories of the taxonomy, allowing us to conclude that FlexConnect is a general-purpose, flexible process modelling language.

## 7.3 Future Work

There are several possible directions for future work. We begin with a discussion of the formalisation presented in Chapter 5. To formalise the semantics of the O-C process modelling approach proposed in this thesis, a CPN was defined that captures the behavior of FlexConnect process models. The "fast-forward" capabilities of the CPN Tools software have been used to test this CPN with example models, by running a large number of transitions and manually inspecting the resulting state of the CPN.

One direction for future work is to extend this CPN in order to enable *stochastic simulation* of FlexConnect process models. Such simulation capabilities would enable analysts to perform cycle-time analysis, resource utilization analysis and activity-based costing of O-C process models. Since the CPN behaves as a process model interpreter (or a workflow engine), there is no need to remodel a new CPN each time a simulation on a different process model is performed. This approach would be appealing for the simulation of process models that may have been changed between simulations since CPN development is well-known as begin quite difficult and time-consuming.

The FlexConnect modelling tool is a proof-of-concept artifact prepared in accordance with the design science methodology. There are many usability improvements that could still be made to the tool outside of the environment it has been applied to over the duration of this project. For example, several improvements could be made to the user interface of the FlexConnect modelling tool, such as automatically adding gateways to states rather than adding them manually, improving the positioning of gateways relative to states, automatically specifying unique IDs for states and gateways and automatically specifying unique names for gateways (which can be done because gateway names are symbolic and often have no real meaning).

The work presented in this thesis on an O-C approach to process modelling is primarily focussed on control-flow. The basic structure of control-flow has been captured, but currently the details of *data* movement and *resource allocation* are missing. Early in the research project, evaluations of the FlowConnect system were completed in terms of the control-flow, data [82], resources [85] and exception handling [84] patterns. These evaluations were done to gain an understanding of how FlowConnect works. In addition to the control-flow pattern support documented in this thesis, the platform was found to support a significant number of resource patterns, as well as a reasonable number of the data patterns and several of the exception handling patterns.

Extending the meta-model proposed in this thesis in order to deal with the resource and the data perspectives is an avenue for future work. Initial work in this direction was undertaken during the early phases of the PhD project, but it was found that a strong control-flow meta-model was needed in order to overlay the data and resource aspects. Accordingly, the meta-model presented in this thesis focuses on the control-flow perspective. Previous research such as that done by Kappel and Schrefl [43, 44] focussed on unifying data and behaviour using object-oriented design techniques. Although based on Petri Nets, these early works are not typically concerned with process modelling. Support for the data perspective in the O-C approach could be achieved by adding the ability to manipulate and exchange data in the ways identified by the Workflow Data Patterns (i.e. data visibility, transfer, interaction and routing).

Similarly, support for the resource perspective could be achieved by primarily adding role types to the O-C meta-model (e.g. user, group, reference, system) and then specifying the privileges that roles can have over activities. Achieving comprehensive support for the Workflow Resource Patterns would require refinement of these basic elements. Inclusion of the data and resource perspectives would then result in a fully-fledged O-C process modelling language.

Another avenue for further research is to investigate process modelling patterns that may be specific to the O-C approach. The motivation driving such an investigation is that the O-C approach brings its own characteristics to process modelling that are not typically found in activity-centric approaches. An initial exploration in this area is found in Künzle *et al* [51].

Reuse in process models is an area that has not received a large amount of attention in the literature. Thus, an additional possibility for future work would be an investigation into the potential for higher levels of reuse of process models. The main benefit of reuse is to increase development efficiency through minimisation of model redesign. Since reuse is a design discipline, it can only be encouraged through the appreciation of best practices and identification of generalisable solutions. However, encouraging greater levels of reuse of process models would have the benefit of avoiding *reinventing the wheel* during a process model development effort, and would also lead towards methods for the rapid development of *mature* process models. The O-C process modelling approach offers an attractive foundation for more reuse-oriented process modelling practices by enabling designers to modularise their process models along key business objects.

An O-C approach requires users to understand and conceptualise business objects. An advantage of activity-centric models in practice is that they are an effective instrument for facilitating communication with business users. Since business users often understand what they do in terms of activities rather than in terms of the objects affected by a process, O-C models can be criticised for being an “unnatural” modelling approach. Communicating in terms of objects may require learning a new process modelling “language”. In practice, this problem can be overcome by increasing familiarity with the O-C paradigm with methods of training such as workshops. Future work on this topic could consist of an empirical investigation to identify the main usability problems of O-C process models.

As mentioned in the Control-flow Patterns Evaluation Summary of Chapter 6, an O-C process modelling language contains some additional modelling complexity compared to an activity-centric language. To capture joins and splits in an O-C process model (e.g. an AND-split) the way of modelling contains a trade-off that *artificial* objects must sometimes be created. This obstacle means it is possible that not all objects in an O-C process model represent an actual object such as an *Inspection* or an *Issue*. However, not every O-C model is affected by this obstacle, which can emerge as a consequence of improperly perceiving a process in terms of objects. The definition of a complexity metric may be helpful to better understand and reduce complexity in O-C process models.

# Bibliography

[1] W. M. P. van der Aalst, M. Adams, A. H. M. ter Hofstede, M. Pesic, and H. Schonenberg. Flexibility as a Service. Technical Report BPM-08-09, BPMcenter.org, 2008.

[2] W. M. P. van der Aalst, P. Barthelmess, C. A. Ellis, and J. Wainer. Proclets: A Framework for Lightweight Interacting Workflow Processes. *International Journal of Cooperative Information Systems*, 10(4):443–481, 2001.

[3] W. M. P. van der Aalst, A. K. A. de Medeiros, and A. J. M. M. Weijters. Genetic Process Mining. In *Applications and Theory of Petri Nets, 26th International Conference (ICATPN 2005)*, pages 48–69, Miami, USA, June 20-25 2005.

[4] W. M. P. van der Aalst, M. Dumas, P. Wohed, and A. H. M. ter Hofstede. Pattern Based Analysis of BPML (and WSCI). Technical Report FIT-TR-2002-05, Faculty of Information Technology, Queensland University of Technology, 2002.

[5] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[6] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[7] W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske. Business Process Management: A Survey. In *Proceedings of the First International Conference on Business Process Management (BPM 2003)*, pages 1–12, Eindhoven, The Netherlands, June 26-27 2003.

[8] W. M. P. van der Aalst, M. Weske, and D. Grünbauer. Case handling: a new paradigm for business process support. *Data and Knowledge Engineering*, 53(2):129–162, 2005.

[9] M. Adams, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA and ODBASE*, pages 291–308, Montpellier, France, October 29 - November 3 2006.

[10] M. Adams, A. H. M. ter Hofstede, W. M. P. van der Aalst, and D. Edmond. Dynamic, Extensible and Context-Aware Exception Handling for Workflows. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS*, pages 95–112, Vilamoura, Portugal, November 25-30 2007.

[11] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. http://dev2dev.bea.com/webservices/BPEL4WS.html, 2003.

[12] K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam, and F. Y. Wu. Artifact-centered operational modeling: Lessons from customer engagements. *IBM Systems Journal*, 46(4):703–721, 2007.

[13] K. Bhattacharya, C. E. Gerede, R. Hull, R. Liu, and J. Su. Towards Formal Analysis of Artifact-Centric Business Process Models. In *Proceedings of the 5th International Conference on Business Process Management (BPM)*, pages 288–304, Brisbane, Australia, September 24-28 2007. Springer.

[14] I. Bider. *State-Oriented Business Process Modeling: Principles, Theory and Practice*. PhD thesis, KTH Royal Institute of Technology, Computer and Systems Sciences, 2002.

[15] I. Bider and M. Khomyakov. Business Process Modeling - Motivation, Requirements, Implementation. In *ECOOP Workshops*, pages 217–218, 1998.

[16] D. P. Bogia and S. M. Kaplan. Flexibility and control for dynamic workflows in the WORLDS environment. In *Proceedings of the Conference on Organizational Computing Systems (COOCS 1995)*, pages 148–159, Milpitas, California, USA, August 13-16 1995.

[17] Y. Bontemps, P. Heymans, and P. Schobbens. From Live Sequence Charts to State Machines and Back: A Guided Tour. *IEEE Transactions on Software Engineering (TSE)*, 31(12):999–1014, 2005.

[18] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley Professional, September 1998.

[19] S. Ellegård Borch and C. Stefansen. On Controlled Flexibility. In *Proceedings of the CAISE*06 Workshop on Business Process Modelling, Development, and Support (BPMDS '06)*, Luxemburg, June 5-9 2006.

[20] J. Campos and J. Merseguer. On the Integration of UML and Petri Nets in Software Development. In *Petri Nets and Other Models of Concurrency (ICATPN 2006), 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency*, pages 19–36, Turku, Finland, 2006.

[21] P. Dadam, M. Reichert, S. Rinderle, M. Jurisch, H. Acker, K. Göser, U. Kreher, and M. Lauer. Towards Truly Flexible and Adaptive Process-Aware Information Systems. In *Information Systems and e-Business Technologies, 2nd International United Information Systems Conference*, pages 72–83, Klagenfurt, Austria, April 22-25 2008.

[22] T. H. Davenport, editor. *Process Innovation: Reengineering Work through Information Technology.* Harvard Business School Press: Boston, MA, 1993.

[23] G. Decker, R. M. Dijkman, M. Dumas, and L. García-Bañuelos. Transforming BPMN Diagrams into YAWL Nets. In *Proceedings of the 6th International Conference on Business Process Management*, pages 386–389, Milan, Italy, September 2-4 2008.

[24] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst. The ProM Framework: A New Era in Process Mining Tool Support. In *26th International Conference on Applications and Theory of Petri Nets (ICATPN)*, pages 444–454, Miami, USA, June 20-25 2005.

[25] D. Dori. *Object-Process Methodology, A Holistic Systems Paradigm.* Springer Verlag, 2002.

[26] M. Dumas and A. H. M. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *4th International Conference on The Unified Modeling Language (UML), Modeling Languages, Concepts and Tools*, pages 76–90, Toronto, Canada, October 2001.

[27] M. Dumas, W. M. P. van der Aalst, and A. H. M. ter Hofstede, editors. *Process-Aware Information Systems.* John Wiley & Sons, New Jersey, 2005.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software.* Addison-Wesley, Boston, MA, USA, 1995.

[29] D. Georgakopoulos, H. Schuster, D. Baker, and A. Cichocki. Managing Escalation of Collaboration Processes in Crisis Mitigation Situations. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, pages 45–56, San Diego, California, USA, 28 February - 3 March 2000.

[30] H. Giese, J. Graf, and G. Wirtz. Modeling Distributed Software Systems with Object Coordination Nets. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'98*, pages 107–116. IEEE Press, 1998.

[31] G. Grossmann, M. Schrefl, and M. Stumptner. Modelling Inter-Process Dependencies with High-Level Business Process Modelling Languages. In *Conceptual Modelling 2008, Fifth Asia-Pacific Conference on Conceptual Modelling (APCCM 2008)*, pages 89–102, Wollongong, New South Wales, Australia, 2008.

[32] T. Halpin. *Information modeling and relational databases: from conceptual analysis to logical design.* Morgan Kaufmann Publishers Inc., 2001.

[33] D. Harel. Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[34] P. Heinl, S. Horn, S. Jablonski, J. Neeb, K. Stein, and M. Teschke. A comprehensive approach to flexibility in workflow management systems. In *Proceedings of the international joint conference on Work activities coordination and collaboration (WACC)*, pages 79–88, San Francisco, California, USA, February 22-25 1999.

[35] R. Hennicker and A. Knapp. Activity-Driven Synthesis of State Machines. In *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007*, pages 87–101, Braga, Portugal, March/April 2007.

[36] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1), 2004.

[37] R. Hull. Artifact-Centric Business Process Models: Brief Survey of Research Results and Challenges. In *On the Move to Meaningful Internet Systems: OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE, Proceedings Part II*, pages 1152–1163, Monterrey, Mexico, November 9-14 2008.

[38] R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative workflows that support easy modification and dynamic browsing. In *Proceedings of the international joint conference on Work Activities Coordination and Collaboration (WACC)*, pages 69–78, San Francisco, California, USA, February 22-25 1999.

[39] IBM Corporation. Business State Machines. `http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/topic/com.ibm.wbit.help.ae.ui.doc/topics/cundstat.html`, 2005.

[40] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Springer-Verlag, 1997.

[41] P. Johannesson and E. Perjons. Design principles for process modelling in enterprise application integration. *Information Systems*, 26(3):165–184, 2001.

[42] G. Kappel, B. Pröll, S. Rausch-Schott, and W. Retschitzegger. TriGS$_{flow}$ Active Object-Oriented Workflow Management. In *Hawaii International Conference on System Sciences (HICSS'95)*, Kihei, Maui, Hawaii, USA, January 3-6 1995.

[43] G. Kappel and M. Schrefl. A Behaviour Integrated Entity-Relationship Approach for the Design of Object-Oriented Databases. In *Proceedings of the 7th International Conference on Entity-Relationship Approach (ER 88)*, pages 311–328, Rome, Italy, November 16-18 1988.

[44] G. Kappel and M. Schrefl. Object/Behavior Diagrams. In *Proceedings of the 7th International Conference on Data Engineering (ICDE 91)*, pages 530–539, Kobe, Japan, 1991.

[45] J. Kim and C. R. Carlson. A Design Methodology for Workflow System Development. In *Databases in Networked Information Systems (DNIS), Second International Workshop*, pages 15–28, Aizu, Japan, December 2002.

[46] A. Kleppe and J. Warmer. Making UML Activity Diagrams Object-Oriented. In *33rd International Conference on Technology of Object-Oriented Languages (TOOLS)*, pages 132–143, Mont-Saint-Michel, France, August 2000.

[47] J. Klingemann. Controlled Flexibility in Workflow Management. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 126–141, Stockholm, Sweden, June 5-9 2000.

[48] P. Kueng, P. Bichler, P. Kawalek, and M. Schrefl. How to compose an object-oriented business process model? In *Proceedings of IFIP TC8, WG8.1/8.2 working conference on method engineering*, pages 94–110, London, UK, 1996. Chapman & Hall, Ltd.

[49] S. Kumaran. The Model-Driven Enterprise. In *Proceedings of the Global EAI (Enterprise Application Integration) Summit 2004*, pages 166–180, Banff, Canada, 2004.

[50] S. Kumaran, R. Liu, and F. Y. Wu. On the Duality of Information-Centric and Activity-Centric Models of Business Processes. In *Advanced Information Systems Engineering, Proceedings of the 20th International Conference (CAiSE 2008)*, pages 32–47, Montpellier, France, 2008.

[51] V. Künzle and M. Reichert. Towards Object-aware Process Management Systems: Issues, Challenges, Benefits. In *10th Workshop on Business Process Modeling, Development, and Support (BPMDS'09)*, Amsterdam, The Netherlands, June 8-9 2009.

[52] J. M. Küster, K. Ryndina, and H. Gall. Generation of Business Process Models for Object Life Cycle Compliance. In *Proceedings of the 5th International Conference on Business Process Management (BPM)*, pages 165–181, Brisbane, Australia, September 24-28 2007.

[53] C. Lakos. Object Oriented Modeling with Object Petri Nets. In *Concurrent Object-Oriented Programming and Petri Nets*, Lecture Notes in Computer Science, pages 1–37. Springer, 2001.

[54] J. Lee, R. Akkiraju, C. Tian, S. Jiang, S. Danturthy, P. Sundhararajan, C. Nordman, R. Mohan, H. Singala, and W. Ding. Business Transformation Workbench: A Practitioner's Tool for Business Transformation. In *2008 IEEE International Conference on Services Computing (SCC 2008)*, pages 81–88, Honolulu, Hawaii, USA, 8-11 July 2008.

[55] F. Leymann and D. Roller. *Production Workflow - Concepts and Techniques.* Prentice-Hall, 2000.

[56] P. Loos and T. Allweyer. Object orientation in business process modelling through applying event driven process chains in UML. In *Proceedings of Second International Workshop on Enterprise Distributed Object Computing (EDOC'98)*, pages 102–112, San Diego, CA, USA, 1998.

[57] R. Mohan, M. A. Cohen, and J. Schiefer. A State Machine Based Approach for a Process Driven Development of Web-Applications. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE 2002)*, pages 52–66, Toronto, Canada, May 27-31 2002.

[58] D. Moldt and R. Valk. Object Oriented Petri Nets in Business Process Modeling. In *Business Process Management*, volume 1806 of *Lecture Notes in Computer Science*, pages 254–273. Springer, 2000.

[59] D. Müller, M. Reichert, and J. Herbst. Data-Driven Modeling and Coordination of Large Process Structures. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA and IS, Proceedings, Part I*, pages 131–149, Vilamoura, Portugal, November 25-30 2007.

[60] D. Müller, M. Reichert, and J. Herbst. A New Paradigm for the Enactment and Dynamic Adaptation of Data-Driven Process Structures. In *Proceedings of the 20th International Conference on Advanced Information Systems Engineering, (CAiSE 2008)*, pages 48–63, Montpellier, France, June 16-20 2008.

[61] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE (invited paper)*, 77(4):541–580, April 1989.

[62] P. Muth, D. Wodtke, J. Weißenfels, A. K. Dittrich, and G. Weikum. From Centralized Workflow Specification to Distributed Workflow Execution. *Journal of Intelligent Information Systems (JIIS)*, 10(2):159–184, March/April 1998.

[63] B. Mutschler, M. Reichert, and J. Bumiller. Unleashing the Effectiveness of Process-Oriented Information Systems: Problem Analysis, Critical Success Factors, and Implications. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 38(3):280–291, 2008.

[64] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.

[65] M. Nüttgens, T. Feld, and V. Zimmermann. Business Process Modeling with EPC and UML: Transformation or Integration? In *UML Workshop*, pages 250–261, Mannheim, Germany, November 1997.

[66] Object Management Group. Business Process Modelling Notation, Ver 1.0. http://www.bpmn.org, 2006.

[67] Object Management Group. Common Object Request Broker Architecture (CORBA). http://www.corba.org/, accessed: January 10, 2007.

[68] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[69] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 287–300, Annapolis, Maryland, USA, 15-19 October 2007.

[70] M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. van der Aalst. Constraint-Based Workflow Models: Change Made Easy. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA and IS, Proceedings, Part I*, pages 77–94, Vilamoura, Portugal, November 25-30 2007.

[71] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell.* O'Reilly Media, 2005.

[72] G. Redding, M. Dumas, A. H. M. ter Hofstede, and A. Iordachescu. SWS FlowConnect - Control Flow Patterns Evaluation. Technical report, WorkflowPatterns.com, 2006.

[73] G. Redding, M. Dumas, A. H. M. ter Hofstede, and A. Iordachescu. Transforming Object-oriented Models to Process-oriented Models. In *Business Process Management Workshops – Proceedings of the 2007 International Workshop on Business Process Design (BPD)*, pages 132–143, Brisbane, Australia, September 24 2007.

[74] G. Redding, M. Dumas, A. H. M. ter Hofstede, and A. Iordachescu. Generating Business Process Models from Object Behaviour Models. *Information Systems Management*, 25(4):319–331, 2008.

[75] G. Redding, M. Dumas, A. H. M. ter Hofstede, and A. Iordachescu. Modelling Flexible Processes with Business Objects. In *11th IEEE Conference on Commerce and Enterprise Computing (CEC 2009)*, Vienna, Austria, July 20-23 2009.

[76] M. Reichert and P. Dadam. ADEPT$_{\text{flex}}$-Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems (JIIS)*, 10(2):93–129, 1998.

[77] M. Reichert, S. Rinderle, and P. Dadam. ADEPT Workflow Management System. In *Business Process Management*, pages 370–379, Eindhoven, The Netherlands, June 26-27 2003.

[78] H. A. Reijers. Workflow Flexibility: The Forlorn Promise (invited paper). In *15th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2006)*, pages 271–272, Manchester, United Kingdom, 26-28 June 2006.

[79] H. A. Reijers, S. Limam, and W. M. P. van der Aalst. Product-Based Workflow Design. *Journal of Management Information Systems*, 20(1):229–262, 2003.

[80] S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems - a survey. *Data and Knowledge Engineering*, 50(1):9–34, 2004.

[81] C. Rolland. A Comprehensive View of Process Engineering. In *Advanced Information Systems Engineering, 10th International Conference (CAiSE'98)*, pages 1–24, Pisa, Italy, June 8-12 1998.

[82] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In *24th International Conference on Conceptual Modeling (ER 2005)*, pages 353–368, Klagenfurt, Austria, October 24-28 2005.

[83] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical Report BPM-06-22, BPMCenter.org, 2006.

[84] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Workflow Exception Patterns. In *Advanced Information Systems Engineering, 18th International Conference, (CAiSE 2006)*, pages 288–302, Luxembourg, Luxembourg, June 5-9 2006.

[85] N. Russell, W. M. P. van der Aalst, A. H. M. ter Hofstede, and D. Edmond. Workflow Resource Patterns. Technical Report BETA Working Paper Series, WP 127, WorkflowPatterns.com, 2004.

[86] N. Russell, W. M. P. van der Aalst, A. H. M. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In *Advanced Information Systems Engineering, 17th International Conference (CAiSE)*, pages 216–232, Porto, Portugal, 2005.

[87] S. W. Sadiq, W. Sadiq, and M. E. Orlowska. Pockets of Flexibility in Workflow Specification. In *Conceptual Modeling - ER 2001, Proceedings of the 20th International Conference on Conceptual Modeling*, pages 513–526, Yokohama, Japan, November 27-30 2001.

[88] O. Saidani and S. Nurcan. A Role-Based Approach for Modeling Flexible Business Processes. In *Proceedings of the CAISE 2006 Workshop on Business Process Modelling, Development, and Support (BPMDS '06)*, Luxemburg, June 5-9 2006.

[89] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W. M. P. van der Aalst. Towards a Taxonomy of Process Flexibility. In *Proceedings of the CAiSE'08 Forum*, pages 81–84, Montpellier, France, 2008.

[90] M. H. Schonenberg, R. S. Mans, N. C. Russell, N. A. Mulyar, and W. M. P. van der Aalst. Towards a Taxonomy of Process Flexibility (Extended Version). Technical Report BPM-07-11, BPMcenter.org, 2007.

[91] M. Schrefl and M. Stumptner. On the Design of Behavior Consistent Specializations of Object Life Cycles in OBD and UML. In *Advances in Object-Oriented Data Modeling*, pages 65–104. MIT Press, Cambridge, Mass., USA, 2000.

[92] M. Schrefl and M. Stumptner. Behavior-consistent specialization of object life cycles. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 11(1):92–148, January 2002.

[93] S. Seidel, F. Müller-Wienbergen, M. Rosemann, and J. Becker. A Conceptual Framework for Information Retrieval in Pockets of Creativity. In *Multikonferenz Wirtschaftsinformatik (MKWI 2008)*, Munich, Germany, February 26-28 2008.

[94] Shared Web Services Pty. Ltd. FlowConnect User Guide. see: SP068 User Guide.doc, August, 2003.

[95] H. Smith and P. Fingar. *Business Process Management: The Third Wave*. Meghan Kiffer Pr, 2003.

[96] M. Snoeck, S. Poelmans, and G. Dedene. An architecture for bridging OO and business process modelling. In *33rd International Conference on Technology of Object-Oriented Languages (TOOLS)*, pages 132–143, Mont-Saint-Michel, France, August 2000.

[97] R. A. Snowdon, B. Warboys, R. M. Greenwood, C. P. Holland, P. J. Kawalek, and D. R. Shaw. On the architecture and form of flexible process support. *Software Process: Improvement and Practice*, 12(1):21–34, 2007.

[98] S. Tolliday and J. Zeitlin. *Between Fordism and Flexibility: The Automobile Industry and Its Workers*. Berg Publishers, 1992.

[99] J. D. Ullman. *Elements of ML Programming*. Prentice-Hall, New Jersey, 1998.

[100] International Telecommunications Union. Formal Description Techniques (FDT) – Message Sequence Chart. Technical Report Z-120, Telecommunication Standardisation Sector of ITU, 2001.

[101] I. T. P. Vanderfeesten, H. A. Reijers, and W. M. P. van der Aalst. Product Based Workflow Support: Dynamic Workflow Execution. In *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE 2008)*, pages 571–574, Montpellier, France, June 16-20 2008.

[102] J. Vanhatalo, H. Völzer, and F. Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In *Fifth International Conference on Service-Oriented Computing (ICSOC)*, pages 43–55, Vienna, Austria, September 17-20 2007.

[103] G. Vossen and M. Weske. The WASA$_2$ object-oriented workflow management system. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data (SIGMOD '99: )*, pages 587–589, New York, NY, USA, June 1-3 1999. ACM Press.

[104] G. Wagner. The Agent-Object-Relationship metamodel: towards a unified view of state and behavior. *Information Systems*, 28(5):475–504, 2003.

[105] B. Weber, S. Rinderle, and M. Reichert. Change Patterns and Change Support Features in Process-Aware Information Systems. In *19th International Conference on Advanced Information Systems Engineering*, pages 574–588, Trondheim, Norway, June 11-15 2007.

[106] I. Weber, J. Hoffmann, J. Mendling, and J. Nitzsche. Towards a Methodology for Semantic Business Process Modeling and Configuration. In *Service-Oriented Computing - ICSOC 2007, International Workshops*, pages 176–187, Vienna, Austria, September 17 2007.

[107] M. Weske. Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System. In *34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, Maui, Hawaii, January 3-6 2001.

[108] R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30(4):459–527, 1998.

[109] G. Wirtz, M. Weske, and H. Giese. The OCoN Approach to Workflow Modeling in Object-Oriented Systems. *Information Systems Frontiers*, 3(3):357–376, 2001.

[110] P. Wohed, E. Perjons, M. Dumas, and A. H. M. ter Hofstede. Pattern Based Analysis of EAI Languages - The Case of the Business Modeling Language. In *Proceedings of the 5th International Conference on Enterprise Information Systems (ICEIS)*, pages 174–184, Angers, France, April 2003.

[111] P. Wohed, N. Russell, A. H. M. ter Hofstede, B. Andersson, and W. M. P. van der Aalst. Open Source Workflow: A Viable Direction for BPM? In *Advanced Information Systems Engineering, 20th International Conference, (CAiSE 2008)*, pages 583–586, Montpellier, France, June 16-20 2008.

[112] P. Wohed, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and N. Russell. Pattern-Based Analysis of the Control-Flow Perspective of UML Activity Diagrams. In *24th International Conference on Conceptual Modeling (ER 2005)*, pages 63–78, Klagenfurt, Austria, October 24-28 2005.

[113] Workflow Management Coalition. Workflow Process Definition Interface - XML Process Definition Language, Version 2.0. http://www.wfmc.org/standards/docs/TC-1025_xpdl_2_2005-10-03.pdf, 2005.

[114] M. T. Wynn, D. Edmond, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Achieving a General, Formal and Decidable Approach to the OR-Join in Workflow Using Reset Nets. In *26th International Conference on Applications and Theory of Petri Nets (ICATPN 2005)*, pages 423–443, Miami, Florida, USA, June 20-25 2005.

[115] M. T. K. Wynn. *Semantics, Verification, and Implementation of Workflows with Cancellation Regions and OR-joins.* PhD thesis, Queensland University of Technology, November 2006.