Realizing a Workflow Engine with the Event Coordination Notation – A Case-study Evaluating the Event Coordination Notation

Jesper Jepsen

Kongens Lyngby 2013
IMM-M.Sc.-2013-101

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013-101

Summary (English)

The Event Coordination Notation (ECNO) is a modelling notation for behaviour modelling, and a technology for generating executable code from these models. Until now, ECNO has only been demonstrated with small example projects. That is our motivation to realize a relatively large application in ECNO. In particular, we have selected to implement a *workflow engine*. The workflow engine is based on AMFIBIA, which is an aspect oriented meta-model for business process models. In AMFIBIA, a so-called core is used to integrate the aspects of business processes, such that no aspect is favoured over others. AMFIBIA demonstrated their concepts in their own workflow engine implementation. Some parts were modelled, but other parts were programmed. A large part of the behaviour was programmed. When we re-implement the ideas of AMFIBIA using ECNO, we can model the most of the behaviour.

The result is a fully working workflow engine, primarily generated from models, demonstrating that ECNO can be used to develop larger applications. A GUI demonstrates the functionality of the workflow engine. We have generated a simple tool for creating business processes that the engine can execute. Based on our experiences from the development of the workflow engine, we have evaluated the conceptual and technical aspects of the current release of ECNO. We conclude overall, that the existing concepts (ECNO release 0.3.1) were sufficient for our purpose, except we had to work around minor issues. The most significant limitation was related to performance. The performance issues were expected, but we have made the problems more concrete, and provided a test application for new ECNO concepts.

Summary (Danish)

Event Coordination Notation (ECNO) er en notation, og en teknologi, til modellering af funktionalitet i software systemer, og til kodegenerering fra disse modeller. Indtil nu, er ECNO kun demonstreret med mindre eksempler. Dette er vores motivation til at prøve med en større applikation i ECNO. Vi har valgt at udvikle en såkaldt *workflow engine*. Arkitekturen i vores workflow engine baserer sig på en aspektorienteret meta-model til modellering af forretningsprocesser, ved navn AMFIBIA. I AMFIBIA bruges en såkaldt kerne til at integrere meta-modeller til modeller af de forskellige aspekter af forretningsprocesser, således at intet aspekt vægtes højere end de andre. Dette giver arkitekturmæssige fordele når systemet skal udvides. AMFBIA har selv demonstreret deres koncepter i en implementation, der er delvist genereret og delvist programmeret. I særdeleshed, er større dele af funktionaliteten programmeret. Vi vil gen-implementere koncepterne fra AMFIBIA, men da vi bruger ECNO, kan vi modellere det meste af funktionaliteten også.

Resultatet er en fuldt fungerende workflow engine genereret fra modeller. En grafisk brugergrænseflade demonstrerer, at vores workflow engine virker. Vi har genereret et værktøj, hvor brugeren kan definere modeller af forretningsprocesser, der senere kan afvikles (enactment) af vores workflow engine. Baseret på erfaringerne fra udviklingen af denne workflow engine, har vi evalueret koncepterne, samt de tekniske aspekter, i den nuværende version af ECNO (0.3.1). Vi konkluderer overordnet, at koncepterne var tilstrækkelige til vores formål, omend vi måtte kompensere for mindre detaljer. Dog savnede vi koncepter til optimering af responstid (performance). At responstiden ikke ville kunne optimeres med denne version af ECNO var ventet, men vi har konkretiseret problemet og stillet en testapplikation til rådighed, til afprøvning af fremtidige løsninger.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

The thesis has a workload of 35 ECTS points.

The indended audience are people with a background in software enginering and software modelling (at M.Sc. level and above).

Lyngby, 06-September-2013

Jesper Jepsen

Acknowledgements

I would like to thank my supervisor Ekkart Kindler for very valuable guidance and advice - and for occasionally breaking his own rule of never repeating himself more than three times.

In addition, I would like to thank Kenneth Geisshirt and Steffen Larsen for reading and commenting on my report.

Contents

Summary (English) i

Summary (Danish) iii

Preface v

Acknowledgements vii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Thesis Objectives . . . 3
- 1.3 Overview of Thesis . . . 4

2 Background 7
- 2.1 Business Process Management . . . 7
- 2.2 The aspects of Business Processes Modelling . . . 10
- 2.3 Example 1: An Error Management Process . . . 10
  - 2.3.1 Background story . . . 10
  - 2.3.2 Identification of tasks . . . 11
  - 2.3.3 Control Aspect . . . 11
  - 2.3.4 Organisation Aspect . . . 11
  - 2.3.5 Information Aspect . . . 12
  - 2.3.6 Discussion . . . 13
- 2.4 AMFIBIA . . . 14
  - 2.4.1 Motivation and Objectives . . . 14
  - 2.4.2 AMFIBIA Meta-Model . . . 15
  - 2.4.3 Aspect synchronisation . . . 20
- 2.5 The Event Coordination Notation . . . 21
  - 2.5.1 A Simple Execution Engine . . . 21

2.5.2 Summary and Advanced Concepts 28
2.6 Related Work 31
2.6.1 Subject-Oriented Programming 31
2.6.2 Aspect-Oriented Programming 32
2.6.3 Process Algebra 33

3 Project Scoping **35**
3.1 Modelling-environment 35
3.1.1 Objectives 36
3.1.2 Limitations 36
3.2 Enactment-environment 36
3.2.1 Objectives 36
3.2.2 Limitations 37
3.3 Database integration (omitted) 37
3.3.1 Objectives 37
3.3.2 Limitations 37

4 Workflow Engine **39**
4.1 Patterns 40
4.1.1 The “instance-of” stereotype 40
4.1.2 The “aspect-of” stereotype 42
4.2 Architectural overview 42
4.3 Realizing the Core 43
4.3.1 Structure of the static model 44
4.3.2 Structure of the dynamic model 46
4.3.3 Global behaviour 47
4.3.4 Local behaviour 49
4.4 Realizing the Control Aspect 52
4.4.1 A formalism independent model 52
4.4.2 An implementation 54
4.5 Realizing the Information Aspect 58
4.5.1 A formalism independent model 58
4.5.2 An implementation 62
4.6 Realizing the Organisation Aspect 67
4.6.1 Analysis 67
4.6.2 Model: Structure 68
4.6.3 Model: Global behaviour 70
4.6.4 Model: Local behaviour 70
4.7 Discussion 70
4.7.1 Task identity 71
4.7.2 Selection of trigger elements in the core 71
4.7.3 Building data structures in actions 72
4.7.4 Duration of events 72
4.7.5 Agents as core concept 72

4.7.6 Instantiation of activities . . . 73
4.7.7 Instantiation of cases . . . 73
4.8 Summary . . . 74

5 Enactment GUI **75**
5.1 ECNO's controller framework . . . 75
5.1.1 Element Event Controllers . . . 76
5.1.2 Engine Controllers . . . 76
5.2 Maintaining GUI lists of interactions . . . 77
5.3 ECNO Connectors . . . 77
5.4 The Enactment GUI . . . 78
5.5 Worklist Viewer . . . 78
5.5.1 The Inbox . . . 78
5.5.2 The Work In Progress . . . 80
5.6 Design and Implementation . . . 81
5.6.1 Design of Worklist Viewer . . . 82
5.6.2 Performance optimization . . . 84
5.6.3 Summary . . . 85

6 Implementation **87**
6.1 Workflow Engine - actual models . . . 87
6.2 Process Definition Tool and Runtime Information . . . 88
6.2.1 EMF editors . . . 88
6.2.2 The goal . . . 89
6.2.3 The problems . . . 90
6.2.4 The solution . . . 90
6.3 The behaviour-state resource . . . 92
6.4 Development workspace . . . 93
6.4.1 Project Structure . . . 93
6.5 Summary . . . 95

7 Acceptance Testing **97**
7.1 Example 2: An Online Book Purchase Process . . . 98
7.2 Building the model . . . 100
7.3 Scenario testing . . . 100
7.3.1 Scenario 1 (case 1): Purchase which goes through . . . 100
7.3.2 Scenario 2 (case 2): Book is unavailable . . . 101
7.3.3 Scenario 3 (case 3): Credit card is rejected . . . 101
7.3.4 Conclusion . . . 102
7.4 GUI testing . . . 106
7.4.1 Test results . . . 106
7.4.2 Conclusion . . . 106

8 Evaluation **109**
8.1 Evaluation of Workflow Engine 109
8.2 Evaluation of ECNO 113
8.3 Performance evaluation 116
8.3.1 Performance of Start and Finish vs. number of cases . . . 117
8.3.2 Performance of Finish vs. number of sessions 119
8.3.3 Performance of Login vs. number of sessions 119
8.4 Summary 120

9 Conclusion **121**

Chapter 1

Introduction

"A programming language is low level when its programs require attention to the irrelevant." - Alan Perlis. Model-Driven Software Development, which deals with software creation from models, has the potential to become the next step up the abstraction ladder in software engineering, but it has not yet caught on in the industry (for instance, refer to [15]). In fact, there are many difficulties and challenges to resolve before software creation from models can become a widely used alternative to programming (see [16]). One of these difficulties is in how the behaviour of a system can be captured in models that integrate well with the structural models [14].

In this thesis, we investigate a proposed notation and technology called the Event Coordination Notation (ECNO) [1], which integrates behaviour models with structural models, and generates executable code from these models. ECNO has already been proven to work with minor example projects, but experiences from bigger software projects is still missing. We will take this step and develop a relatively large application with ECNO. As an example, we have chosen a workflow engine. As we will elaborate on later, the choice of a workflow engine has reasons that relates to the history of ECNO.

1.1 Motivation

When using software modelling, for instance, UML diagrams [1], in the development process, developers can start out at a high level of abstraction and ignore implementation and platform details. The focus can then be kept on the more interesting parts such as domain concepts, business logic and use cases. Furthermore, a graphical notation (as in UML) is often preferable over pure code for discussions with customers, and for documentation purposes. However, despite of models often being a good starting point, they require a significant effort to keep them synchronized with the code base after the implementation of the system has started. When the model and the implementation, over time, are no longer reflecting each other, it is tempting to abandon the model, leaving the code as possibly the only remaining updated "design document". In other words, a model which was created in the analysis phase is at high risk of becoming obsolete later on.

Trying to overcome this issue, and just as importantly, with an aim to increase overall productivity by raising the level of abstraction, a large amount of research has gone into code generation from models (Model-Driven Software Development). Then the models effectively becomes part of the "source code". However, the existing tools for model based code generation often focus more on the structural part of the design, and less on the functionality. While it is also technically possible to create behavioural (executable) models and generate code from these models, the integration with the structural models is lacking in current approaches (according to Kindler in [1]). In an earlier publication [14], Kindler writes in the abstract that: "the vision of model-based software engineering is to make models the main focus of software development and to automatically generate software from these models. Part of that idea works already today. But, there are still difficulties when it comes to behaviour. Actually, there is no lack in models for behaviour, but a lack of concepts for integrating them with the other models and with existing code."

The Event Coordination Notation proposed by Kindler [1] addresses the challenge of integrating structural models and behavioural models. ECNO is a notation for behaviour modelling on top of UML class diagrams, a code generator taking these models as input, and an execution engine running the generated code. Combined, ECNO allows us to generate executable code directly from the models. The behaviour models are based on the concepts *events* and *coordinations* defined in a so-called *coordination digram*. The coordination diagram is basically just a UML class diagram with these new concepts added on top of it, and this is the main reason why ECNOs behaviour models integrates so well

[1] www.uml.org

with traditional structural models.

The concepts and technical aspects of ECNO will be explained in greater detail in Section 2.5, but now we move on and define the objectives of this thesis.

1.2 Thesis Objectives

It has been shown that ECNO can generate executable code for a program that simulates a coffee brewer. Actually, a few more ECNO examples exist. One of these implements the firing rule of Petri nets [11]. Another simulates workers carrying out jobs. These examples are supported by a limited and generic GUI which enables the user to interact with the simulation. The problem is, that these examples are still very simple, and we need to find out whether this works for bigger applications. We have therefore selected to perform a case study in which we realize a workflow engine. A workflow engine is an IT system, which supports management and enactment of business processes. We will argue for the choice of application in this section, but first we will list the overall objectives of the case study. Note that the requirements to the workflow engine are in Chapter 3.

- Identify limitations in ECNO when we use it to develop a larger application. It is expected beforehand, that the latest release of ECNO (in the time frame of this thesis, version 0.3.1) will have significant limitations when taking this step.
- Produce an application that demonstrates the capabilities of ECNO version 0.3.1.
- Collect experiences, which in the future can be taken as input in the development of a methodology for ECNO. In other words, provide information and practises on the usage of ECNO.

In the following we will ague for the choice of implementing a workflow engine as a method to achieve these objectives. The choice was slightly biased because ECNO has roots in business process modelling. In particular, ECNO's concepts are generalisations of aspect synchronisation concepts developed for an aspect oriented meta-model and execution engine for business process models. This work is known under the name of AMFIBIA [2]. We will base our work on AMFIBIA, which means we don't have to start from scratch, when we develop the models of the workflow engine. This will also serve another purpose, in

particular, to show that the ideas in AMFIBIA can be realized more easily with ECNO. Therefore, this is not yet a case study for applications in general, but a first step towards it. The task of proving ECNO in a domain other than its historical origin would be an interesting continuation, which is beyond the scope of this thesis.

It should be mentioned up-front, that we do not expect this workflow engine will be able to compete with the available industrial workflow engines, this would simply be too ambitious for a master thesis, not to mention when using a technology, which is at the prototype level. In addition, the workflow engine is not meant to be used by real end users, it is created as a demonstration and testing application for ECNO. Therefore, we can allow to omit features found in all real workflow engines, such as a real database integration, distributed computing and interfaces to external applications.

In terms of contribution, this thesis aims to advance current knowledge by combining existing concepts in ECNO with those of AMFIBIA within a single application, and by creating the first "almost-real" application based on ECNO - a workflow engine.

1.3 Overview of Thesis

This thesis is structured as follows.

Chapter 2: Background
We will initially introduce Business Process Management (BPM) and business process modelling. Then we introduce AMFIBIA followed by an introduction to ECNO. At last we present related work.

Chapter 3: Project Scope
Here we will explain the details of the goals and limitations of this thesis.

Chapter 4: Workflow Engine
Contains the conceptual contribution of this thesis. In particular, the models from which we can generate a workflow engine with ECNO.

Chapter 5: Enactment GUI
This chapter discusses the design and the implementation of an enactment GUI for the end user. The main purpose of the GUI is to demonstrate that the workflow engine is working.

Chapter 6: Implementation
Addresses the implementation concerns for the project overall, which were not covered in Chapter 4 and 5.

Chapter 7: Acceptance Testing
Here we will do acceptance testing of the workflow engine based on a business process example.

Chapter 8: Evaluation
We will here evaluate the workflow engine and ECNO respectively.

Chapter 9: Conclusion
The final conclusion of this thesis.

Chapter 2

Background

This section will introduce the reader to Business Process Management (BPM) and to the basics of business process modelling. It will then describe the work, which this thesis builds on, namely AMFIBIA and ECNO. Finally we will summarise related work. With that, a number of terms and concepts required to understand the main contribution of this thesis will be explained. The vocabulary in this report follows the ones developed in AMFIBIA and ECNO.

2.1 Business Process Management

In this section we give a brief introduction to Business Process Management (BPM) and to BPM software. We are inspired mainly by the work in AMFIBIA. The authors of AMFIBIA are inspired by the work of Wil van der Aalst [6], Leymann and Roller [7], and by the standard in the Workflow Management Coalition [1].

A business process may be defined as a collection of tasks, which may be ordered or partially ordered, and which are acted out by resources within a company to reach a specific goal. As an example, a company may have an equipment

[1] www.wfmc.org

purchasing process involving selection of items, placement of requests, approval of requests, payment, and so forth. Such a process could serve an internal goal of owning that piece of equipment. Wikipedia [2] divides business processes into Mangement processes, Operational processes and Supporting processes. The example above would belong to the category of Operational processes.

Business Process Management could be described as the management process of designing, implementing, enacting and continuously improving business processes. In principle, BPM has nothing to do with computer science (it is a topic of business management), but today, most companies are using some form of IT-support to manage their business processes.

The Workflow Management Coalition (WfMC) is an organisation founded in 1993 with the aim to develop standards for business processes, and to educate the market in "related issues". In what they call "The Workflow Reference Manual" [10], they define *workflow* as "the computerised facilitation or automation of a business process, in whole or part." They further define a *workflow management system* as "a system that completely defines, manages and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic".

Figure 2.1 shows how WfMC defines the components of a workflow management system. As can be seen, a *definition tool* generates a *process definition*, which a *workflow engine* can interpret and execute. The process definition and the engine may both refer to an organisation model. Tasks that are relevant to a particular user are shown in a *worklist*, that is visible to that user in a GUI. When needed, the engine invokes external applications (such as word processing tool, spread sheets etc., and the data is stored in a database. As indicated, a part of this data is workflow relevant and directly used by the engine in the process execution. For instence, this might be a data field to indicate a document is approved or not.

This chapter will continue by focussing on the process definitions, which we also call *business process models*. The act of creating and maintaining these models, would be referred to as doing business process modelling. Understanding of the nature of the business process models is the key to creating a meta-model, which is what AMFIBIA does. First, we will explain how business processes can be divided into aspects.

[2] http://en.wikipedia.org/wiki/Business_process

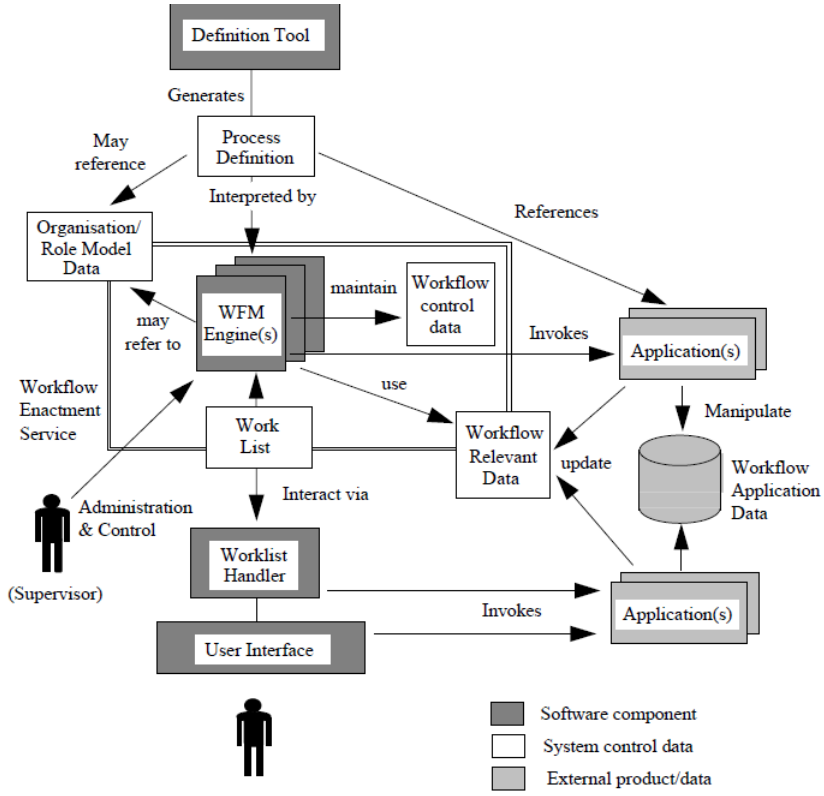

Figure 2.1: Image from The Workflow Reference Model (www.wfmc.org).

2.2 The aspects of Business Processes Modelling

A *business process model* specifies the artefacts of the process, such as tasks and documents, and defines how actors (human or machine resources) interact with the artefacts to complete the process goal.

Although the emphasis may be on a single or two aspects, it is commonly accepted in the literature that there are three main aspects to consider when modelling business processes. Those are the aspects of organisation, control and information (data). Very briefly, the organisation aspect defines organisational structure and resources, the control aspect defines task order, and the information aspect defines the data of the process. There are other aspects, such as transaction and authentication, but they are not discussed in this thesis. To avoid repetition, the main aspects will be further elaborated on in the section on AMFIBIA instead of here (2.4).

2.3 Example 1: An Error Management Process

At this point, we will give a concrete example of how a business process can be defined using aspect oriented modelling. The example will initially serve as foundation for understanding the AMFIBIA meta-models, but it will also serve as a "running example" throughout this thesis.

2.3.1 Background story

An imaginary software company had released a new product but, unfortunately, the product still contained a number of errors being reported by angry customers. But sometimes, the error reports were not descriptive enough to understand the problem. To make matters worse, the company did not have a system for routing errors to the right people, and they had difficulties tracking what happened to the reported errors after they were received. So they decided to formalise a business process for their error handling, thinking they would afterwards find a suitable workflow system to help them implement the process. They broke down the problem to that of identifying the tasks, and then respectively defined the control-, the organisation- and the information aspects. The outcome of this work follows.

2.3.2 Identification of tasks

Obviously there is a task for creating errors (Submit). It is further known that some errors are ignored (Ignore). The company is often fixing errors in the graphical user interface (Correct GUI). Otherwise, the error is usually related to the database implementation (Correct DB). To make sure errors are routed correctly, they also need to be filtered at some point (Filter). Sometimes an error needs to be clarified further (Clarify).

2.3.3 Control Aspect

The Submit task initiates the process. The next task to execute is Filter. After the error has been filtered, one of the tasks Correct GUI, Correct DB, Ignore, or Clarify shall execute. Clarify loops the control back to Filter after execution. There are several notations to express this ordering of tasks, the company used a Petri net, which can be seen in Fig. 2.2. We assume the reader to be familiar with Petri net notation and semantics.

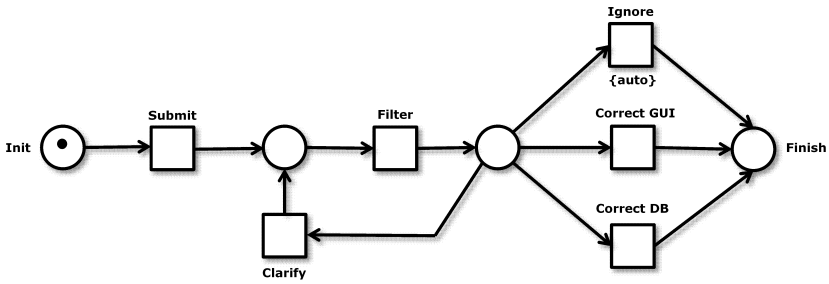

Figure 2.2: The *control aspect* of an error handling business process.

2.3.4 Organisation Aspect

The rather small company has only 3 roles. Those are Manager, GUI Programmer and Database Programmer. Apart from this, the role of Custumer is defined. The tasks Submit and Clarify can be taken by the Customer. There is a restriction that Clarify can only be taken by the person who submitted the error earlier. This is modelled with a *follows up on* relation from Clarify to Submit. Only a Manager may perform the task Filter. A GUI Programmer

can perform the task Correct GUI, and a Database Programmer can perform the task Correct DB. Jack is a Customer, Simon is hired as manager, Paul is a GUI Programmer and Tim is as Database Programmer. Refer to Fig. 2.3 for an illustration.

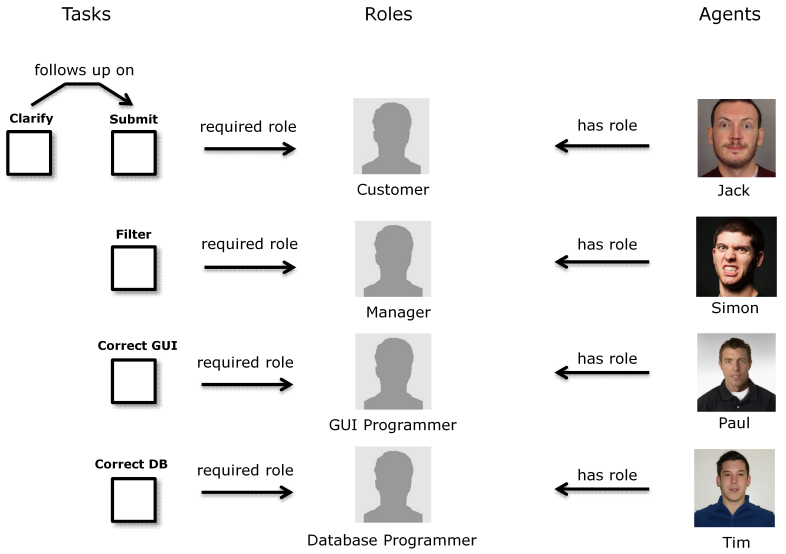

Figure 2.3: The *organisation aspect* of an error handling business process.

2.3.5 Information Aspect

The task Submit does not require any input documents, however, it outputs one document named *error_report*. Filter takes *error_report* as input document and cannot start unless this document exists. Filter also outputs *error_report*. The tasks Ignore, Correct UI, Correct DB, and Clarify all ask for the same input document - e.g *error_report*. In addition, they define start conditions saying that the document field *decision* is equal to one of the values "clf", "ign", "ui", or "db" respectively. This allows the manger handling the Filter task to guide the process based on the value he assigns to this field. These conditions are indicated in the models by dotted arrows with a label stating the condition. Correct UI and Correct DB both outputs the process document *correction_report* on their termination. The task Clarify outputs *error_report*, hopefully in a clarified state.

Still, we do not know what the documents contains, except we have mentioned

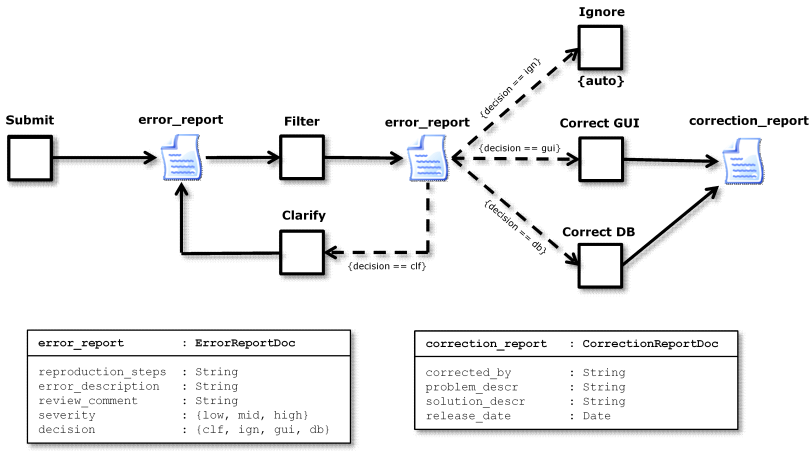

Figure 2.4: The *information aspect* of an error handling business process.

a field named *decision*. However we might guess that *error_report* contains fields for describing the nature of an error. In particular, this document has the type ErrorReportDoc, which define fields for stating how to reproduce the error, for giving an error description, for writing a review comment, for defining the severity, and for making a decision (the latter two are enumerations). The document *correction_report* has the type CorrectionReportDoc, which defines fields to hold information about the problem, information about the solution, the correction release date and so forth. Refer to Fig. 2.4 for an illustration of the information aspect, the used documents and their types.

Until now we only mentioned one type of condition. This was a document condition on the *decision* field in *error_report*. Actually, to ensure a successfull process execution, additional conditions would be of help. For example, the Submit task could have finish condition saying that the *error_description* field cannot be empty. The Filter task could have finish condition stating that *decision* must set to one of “clf”, “ign”, “ui” or “db”.

2.3.6 Discussion

This completes the example. We will now add a few more words to the concepts of input and output documents and how they affect a task execution. If a required input document does not exist (or has unsatisfied start conditions), the task is prevented from starting. If a required output document does not

exist (or has unsatisfied finish conditions), the task is prevented from finishing. If an output document does exist when a task is started, the document will be presented in the task. If it does not exist, it shall be possible to create a new output document from within the task, or to point to an existing document to use.

Due to their visual similarity, the reader might wonder if the information aspect makes the control aspect somewhat redundant. Actually, that is not the case. The information model does not "care" about the ordering of tasks, it only defines which documents (and perhaps which document conditions) are required to start and finish a task. For the sake of argument, let's pretend the control aspect was omitted. First of all, it would now be possible to execute the Submit task multiple times. Second, if the Customer sets the decision field to the value of "gui", it would be possible to by-pass the Filter task, and start a Correct GUI task without involving the Manager. Note that there exists concepts for controlling a process execution using data only. These are used in systems, which do not have an explicit control model, but we will not discuss that here.

2.4 AMFIBIA

While this thesis directly builds on the results of the AMFIBIA project [2], we will summarise their work in this section.

2.4.1 Motivation and Objectives

The AMFIBIA project is motivated by what the authors view as a lack of consistency in the perception of what a business process model really is. They argue that while it is recognized in literature that the aspects and concepts of organisation, control and information are the three most important, "the concrete formalisms, notations, and, in particular, the business process modelling tools vary and are not compatible to each other" (p. 2). The article points out that many formalisms and tools have a build-in bias towards a single aspect or a single formalism making it difficult to add new formalisms or aspects. According to the authors, these problems should be addressed before the problem of defining common exchange formats for data and process models. The authors summarise their objectives as (p. 3):

- "It should cover all basic aspects of business process models and should not be biased toward or focused on one of these aspects."

- "It should be open so that other aspects can be easily added and integrated to it."
- "In particular, there should be clear interfaces for the different aspects and a mechanism for their integration."
- "It should be independent of a particular formalism or notation for business process models. But, it should be easily possible to map existing business process modelling notations to it."

2.4.2 AMFIBIA Meta-Model

AMFIBIA is an aspect oriented meta-model for business process models which does not favour any particular aspect, formalism, or notation. Its elements are explained below. Most of the concepts in this meta-model can be related directly to Example 1.

2.4.2.1 The Core

First of all, the concept of having a *core* is introduced. The purpose of the core is to have something neutral with which the aspects can be integrated.

The core meta-model can be seen in Fig. 2.5. It contains the business process modelling concepts that are commons to all aspects; to be specific these are defined to be *processes* containing *tasks*. Please note that the core concept process is really meant in a general sense of *business process* and should not be confused with the not yet explained *control aspect* concerned only with the ordering of tasks. A task is the blueprint of a specific piece of work.

The term *case* is used to refer to a specific instance of a process. Likewise, *activity* is an instance of a task. Process, task, case and activity are all part of the core model, however AMFIBA makes a distinction between *modelling concepts* (alternatively: static model) and *instance concepts* (alternatively: dynamic model, or runtime model). Case and activity are not in the business process models but are essential for implementing a runtime environment for their execution. Both of these categories of concepts exists side-by-side in the same model using an "instance-of" relation in-between. In the core meta-model, as well as in the aspect meta-models, it is only allowed to have reference from the dynamic model to the static model, never in the other direction. It would be almost impossible to manage, if a static object had to maintain a list of all their dynamically allocated instances, and it would also be conceptually wrong.

In the section about design patters (Sect. 4.1) we will elaborate on the meaning of the "instance-of" stereotype and relate it to technical instantiation.

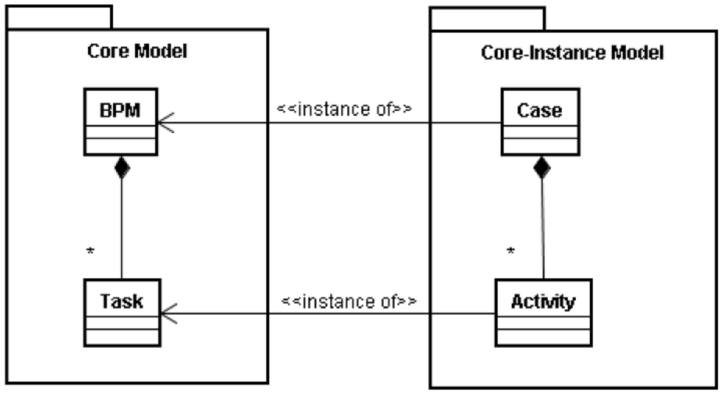

Figure 2.5: Meta-model for the Core (image from: [2]). Recall that BPM refers to the concept process in the text.

2.4.2.2 Aspect integration

For adding *aspects* to the core, AMFIBA defines an "aspect of" relation. Quoting the article, "the meaning of the stereotype aspect is, that there is a relation to an element in the core". For example an aspect is likely to have an element task which is an aspect of task in the core. The task of the aspect will talk about the same task, but from the point of view of the aspect. Which aspect a specific task element belongs to is identified by having a unique package name for each aspect. The meta-model for aspect integration can be seen in Fig. 2.6.

AMFIBIA proposes a mechanism for synchronizing the aspects with the core (and possibly with each other although it it not demonstrated) but that will be explained later. We will now turn the attention to the meta-models for the three main aspects of business process modelling as described in AMFIBIA.

2.4.2.3 Control aspect

The meta-model in Fig. 2.7 defines, in the control aspect, the concepts known from the core: process, case, task and activity. It further defines the *state* concept which is attached to case. The state holds the information of which

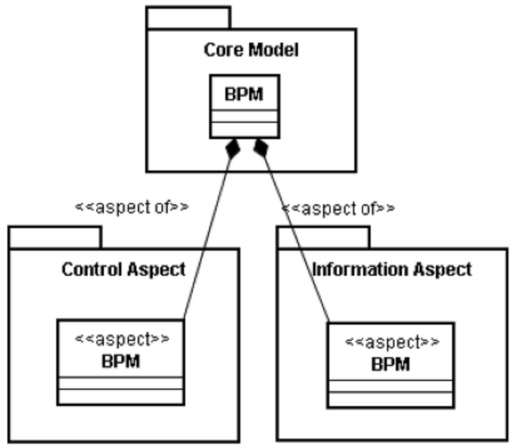

Figure 2.6: The AMFIBIA concepts for aspect integration (image from: [2]).

tasks are *activated* in a given state, as indicated by the reference from state to task. An activated task is defined as one that can be started in a given state. The task implements the two operations *initialize(..)* and *finalize(..)*, both takes the current state as input and returns a new state. *Initialize* returns the state that results from starting a task, while *finalize* returns the state that results from finishing a task. With this, the order (or partial order) in which tasks can be executed is expressed in terms of states and state transitions. In general, tasks may execute in sequence or concurrently.

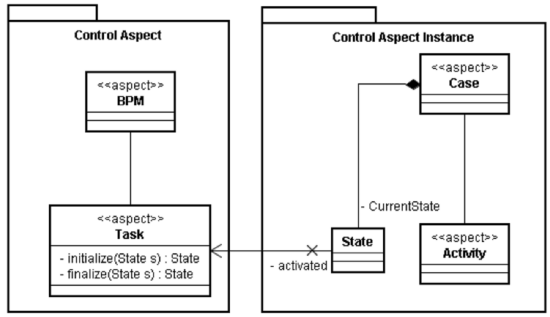

Figure 2.7: Meta-model for the Control Aspect (image: [2]).

The authors of AMFIBIA have generalized their meta-models, assuming no specific formalism or notation. But they do give examples of an implementation

of the control aspect. See in Fig. 2.8 how they implement the control aspect with Petri nets. Notice how tasks are implemented by transitions and how state is implemented by a Petri net marking. We say that the model in Fig. 2.7 is formalism independent and that the implementation in Fig. 2.8 plugs in a formalism dependant model.

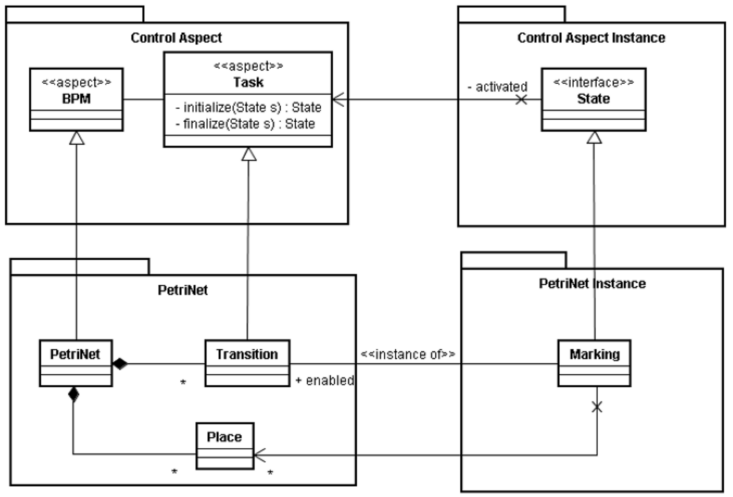

Figure 2.8: Meta-model for a Petri net implementation of the Control Aspect (image: [2]).

2.4.2.4 Information aspect

The meta-model for the information aspect is replicated in Fig. 2.9, where the main feature is that tasks can have input and output *document descriptors*. A document descriptor returns with *selectDocument(..)* the appropriate document given a context (a case). The rules for getting the right document is an implementation issue that the formalism independent model does not define. However, as indicated in the model, the document descriptor must use the concept *document type*. The model also defines that input and output *documents* are attached to activities at runtime, and that a given document is a conceptual instance of a given document type. The model also defines relations between documents types, and these are called links when they are instantiated and refers to documents. Furthermore, document types may be atomic or complex, where the latter kind can contain several documents.

AMFIBIA proposes an implementation model, based on a relational database,

implementing document descriptors by SQL queries and documents by query results. The figure is omitted from this report, but the principle of plugging in a formalism dependant model is the same as shown in the control aspect.

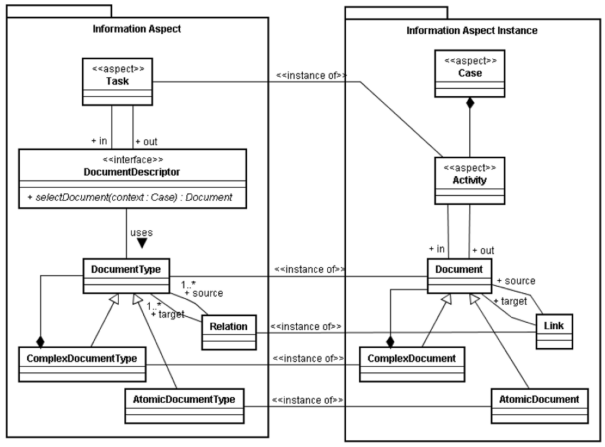

Figure 2.9: Meta-model for the Information Aspect (image: [2]).

2.4.2.5 Organisation aspect

AMFIBIA explains the concepts but omits the meta-model for the organisation aspect with the comment that is follows the same concepts as already demonstrated by the other meta-models. So here, we will just briefly identify the main concepts from Example 1, and otherwise refer to the main section of this thesis (Chapter 4) where a meta-model is developed.

The *organisation* aspect defines the structure of the organisation where the business process is taking place. An organisation model, which may be divided into units and groups (not in Example 1), defines resources and roles. Human resources are called agents. Resources and agents change frequently, and are therefore not part of the static organisation model. Per definition they belong in the *runtime model*. The main purpose of the organisation model is to define which resources a task could, potentially, be assigned to at runtime. AMFIBIA mentions that the behaviour could be captured using so-called *resource descriptors* for tasks, which resembles document descriptors.

2.4.3 Aspect synchronisation

As explained, each aspect has process and task elements which are aspects of elements in the core. We have seen this relationship was modelled structurally using compositions. AMFIBIA models the behavioural part of this relation using automata. Automata for a case element in the core, and in the control aspect can be seen in Fig. 2.10.

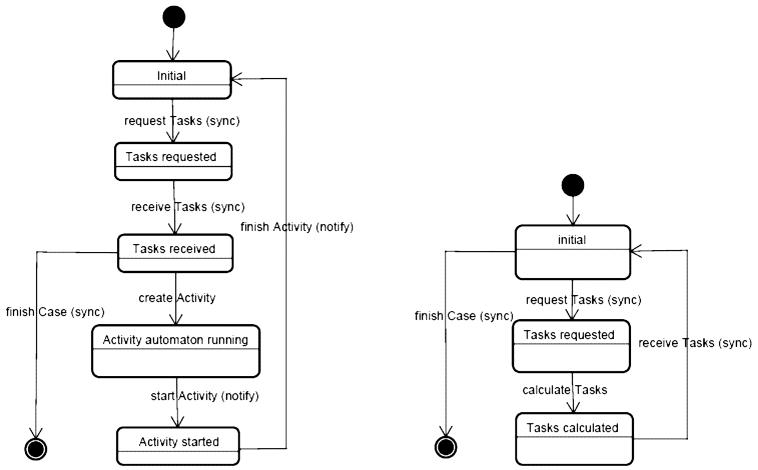

Figure 2.10: Automatas for case in the core (left) and for case in the control aspect (image: [2]).

The automata models consists of states and transitions, where the latter are coupled to *events*. AMFIBIA defines events as "the points of a case execution, which are especially interesting in respect to aspect coordination or to the functionality of the core". Specifically the events are named: "start Case", "finish Case", "request Tasks", "receive Tasks", "create Activity" and "finish Activity". The keyword *sync* means that the event must synchronize with other aspects at the points in their execution where the same event name (and the sync keyword) occurs respectively. The keyword *notify* is used for synchronization of elements within the same aspect. In particular an activity must notify its owner case when it has finished executing.

From the initial state, the "Request Task" event triggers in each aspect a calculation of activated tasks from the point of view of each aspect, and meanwhile the core waits in the state "Tasks requested". The "Receive Task" event is then used to communicate the results back to the core. The core can then create

activities for tasks that are activated in all aspects, and so forth.

Interestingly, the concepts for event coordination in ECNO, are inspired by event synchronisation in AMFIBIA.

2.5 The Event Coordination Notation

In this section, we will explain the key concepts in the Event Coordination Notation. To make it concrete, we will use a small example project implemented within the scope of this thesis. We cannot use Example 1 here, because it would be too complicated for this purpose. The example project used in this section, can be viewed in isolated from the workflow engine, which we will present later.

After the example, we will summarize the concepts, and further introduce some more advanced concepts.

2.5.1 A Simple Execution Engine

This example consists of a simple business process and a simple execution engine, which can execute this particular business process (it is build in). The execution engine comes with a simple GUI for the end user. We will first present the business process, then we present the GUI, before we present the models that realize the execution engine.

2.5.1.1 Process definition

We define a very simple process model consisting of two tasks. Those tasks are named “activity 1” and “activity 2” and they follow each other in that order. The tasks can only be assigned to agents that have the right role. Respectively, they can be assigned to the roles Engineering Manager and CEO. We don’t care about what these tasks really do, it is not important. Using the same graphical notation as earlier, Fig. 2.11 and Fig. 2.12 represent the process described above.

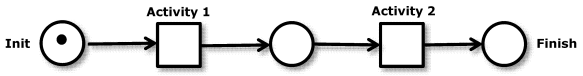

Figure 2.11: Control aspect of the build-in process.

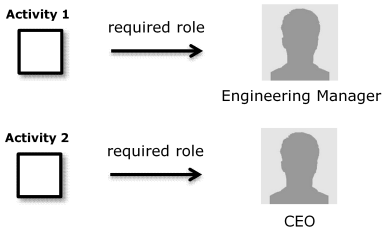

Figure 2.12: Organisation aspect of the build-in process.

2.5.1.2 The GUI - in action

Figure 2.13 shows a screen shot of the active GUI. In the leftmost GUI frame, a user can enter a client name, and then press the button to start a new case for that client. Active cases, are displayed in a list view, showing the case id and the client name. In the rightmost GUI frame, "agents" may log in and take a role. When logged in, an agent can press the button "Push..." to get a random enabled activity matching his role into his *worklist*.

Note that, we use the term *push* a bit inaccurately. In BPM, the terms *pull* and *push* are referring to the way a task is assigned to an agent. Very briefly, when pulling a task, an agent selects a task, which he then receives. When a tasks is pushed to him, he just receives it whether he wants it or not. So, in our case, it would be more right if someone else than the agent pressed that button.

Moving one, the "Push..." button will automatically disable when there are no more activities, in any active cases, that matches the agents role. The received activities are listen in the worklist view, and the agent may select one and execute it at any time. When the two activities corresponding to the two tasks in our fixed process have been executed, the case will change status to "finished" and will disappear from the active cases list in the first GUI window.

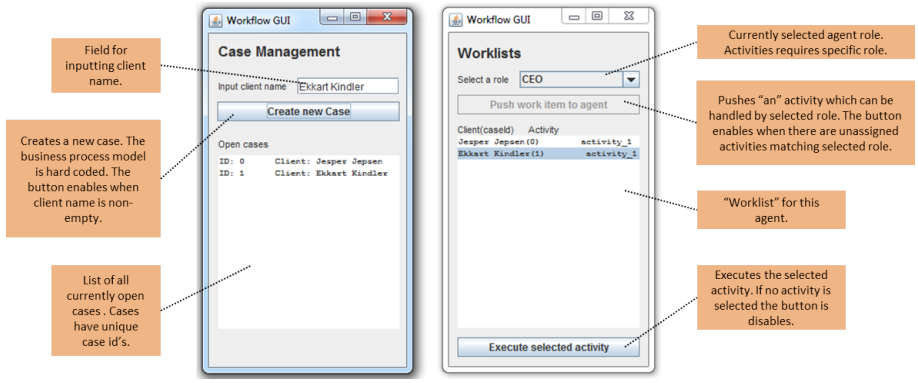

Figure 2.13: Simple Case Management Tool.

2.5.1.3 Execution Engine: Introduction

We now present the ECNO realisation of a selected part of the implementation, starting with the structural model, which is just a class diagram (Eclipse Ecore format), and followed by behavioural models.

ECNO uses two kinds of behavioural models. One for defining global behaviour, and one for defining local behaviour. Global behaviour defines *events* and *coordinations*, in a *coordination diagram*, on top of a class diagram. Local behaviour defines the internal behaviour of the classes (actually, *element types*) with *ECNO nets*.

2.5.1.4 Execution Engine: Structural model

A selected part of the structural model of execution engine is shown in Fig. 2.14, in the form of a conventional class diagram. The model expresses that a *case manager* contains *cases*, and each case contains *activities*. The case refers to *process* representing the build in business process. Specifically, the business process behaviour will be defined by local behaviour of process. While the engine only has a single build-in process, there are no explicit *tasks* in this meta-model. Recall that tasks are modelling concepts and activities are instance (runtime) concepts.

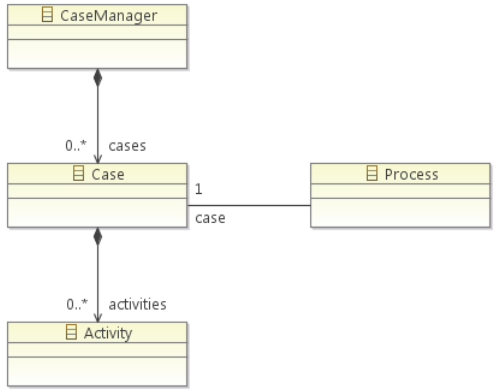

Figure 2.14: Structural model (not complete).

2.5.1.5 Execution Engine: Global behavioural model

We will now continue by talking about the behaviour. We will focus on the behaviour for creating new activities based on the build-in model. Please refer to Fig. 2.15 showing the relevant part of the coordination diagram. Note that we use an ad-hoc notation in the figure, which has same level of abstraction as the real model (it just looks a bit nicer).

The global behaviour model for creating activities involves two classes from the structural model: *process* and *case*. *Process* is involved because it knowns which activities can be created at a given moment. *Case* is involved because it shall own the new activity. For our purpose, we define an *event type* named CreateActivity in a coordination diagram. In same diagram, we have the two involved classes, but since this is a coordination diagram, the classes are called *element types* instead of classes.

By writing the name of the event type, CreateActivity, inside the element types of process and case, we are stating that they are capable of *participating* in an event of that type, for what concerns the global behaviour. Note that, as we will see later, the local behaviour of each element respectively, will further restrict when they can participate. The event type CreateActivity is also annotated on the reference going from process to case. This defines a *coordination*, saying that if process is going to participate in CreateActivity, then it must have a *link* to a (ONE) case, which also can participate. At last, see that our event type CreateActivity is taking a single *event parameter* named "newActivity", of type activity. This parameter makes it possible for the two elements to communicate

about an activity element, we will clarify this in the following section.

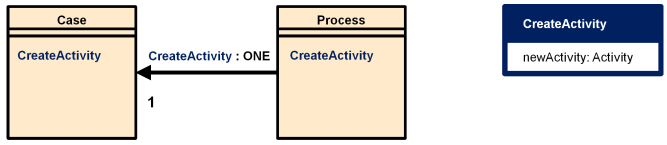

Figure 2.15: Global behaviour for creating activities.

2.5.1.6 Execution Engine: Local behavioural model

The local behaviour of element types defines when they from a local point of view can participate in events, and what happens when they do. For this purpose, we use the so-called *ECNO nets*. ECNO nets are a variation of traditional Petri nets, where some new concepts are added. These concepts are making it possible to *bind* event types, *conditions*, and *actions* to (Petri net) transitions. These concepts works as follows in the example project.

In case, we create a single free floating transition with an *event binding* to CreateActivity (see Fig. 2.16). The keyword *none* expresses there is a single parameter in the signature of the event type, and that case will not pass a value to it. In Petri nets, a free floating transition is always enabled, and since we're not binding any conditions here, case can locally always perform CreateActivity. In the figure, the two lines of Java code below the transition is an *action binding*. This code will be executed when the transition fires. The first line of code uses the ECNO keyword *self* to access the getter for the activity list in case (refer to the structure model). The second line adds an activity to the list. The activity it adds is the one in the event parameter, which is accessed though the local event variable *c*. We will see shortly, in the local behaviour of the process, how this parameter value is passed. The second line of code just sets a backward reference - a bit excessive, and could have been avoided with "opposites" in EMF.

We continue by explaining the local behaviour of a process as shown in Fig. 2.17. Unlike in case, the transitions in this ECNO net have input (Petri net) places. Initially, there is a token in the upper left place enabling the first of two transitions, which has an event binding with CreateAcivity. At the position of the first and only parameter of the event, we call a factory method *createActivity(..)* creating new activity object (element in ECNO terms), while specifying the activity name, "activity1", in the first of three parameters accepted by the factory method. The other two parameters can be ignored for now.

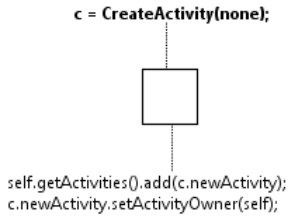

Figure 2.16: Local behaviour of *case*.

After the first transitions fires (we will clarify later how this happens) the next transition becomes enabled. It binds with a different event type, ExecuteActivity, which allows the execution of the activity, which was just created. We can ignore ExecuteActivity, since we focus on the creation of activities in this storyline. After the second transition has fired the same pattern repeats, but this time for the activity named "activity2".

Actually, creating an activity element in the parameter list is quite expensive, because ECNO creates them every time it evaluates the event. A better solution here would have been passing the string name of the activity, and creating it in the action binding of case, when the transition fires.

2.5.1.7 Interactions

Now we have finished explaining the models and we can put everything together by imaging a scenario at runtime. Note first that, *event* is a term for an instance of an event type. Likewise, *element* is an instance of an element type. Elements are to element types what objects are to classes - this is easy to accept. That we can also instantiate event types, and get events out of that, is something that requires a little more getting used to. However, it is easy to accept that the same kind of event can occur many times in a system during runtime, and ECNO just uses event type instantiation to enable this.

Assume now a situation at runtime, where we got one process element with a token in the initial place, and the process element has one linked case element. Then we say that the global and local pre-conditions to *trigger* a CreateActivity event in process are met. In a situation such as this, the ECNO execution engine can, on request, find a so-called *interaction* for CreateActivity, in the element process. The interaction represents the fact that a given behaviour can

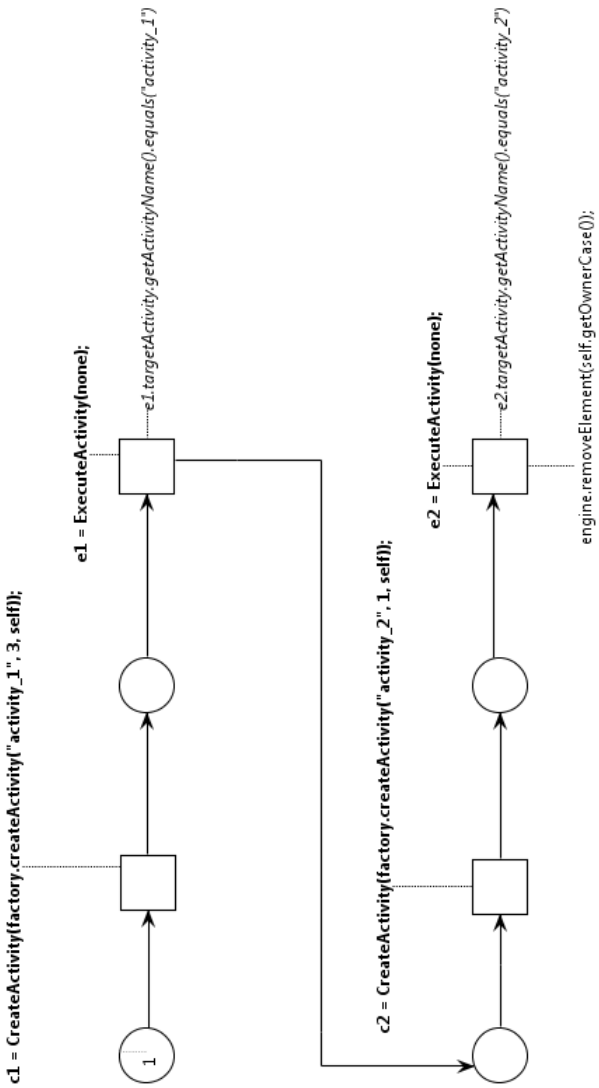

Figure 2.17: Local behaviour of *process*.

be executed. When this particular interaction, of our example, is executed, the effect is that a new activity element will be created and attached to the case. We will not explain here how the new activity becomes visible in the GUI. In Chapter 5, we explain how *engine controllers* are used for this purpose.

The ECNO engine does not execute interactions itself, that is a responsibly of the application. For this purpose the ECNO engine exposes so-called *element event controllers*, which allows applications to "hook in" to the behaviour of elements, get the possible the interactions, and execute them, if wanted. Applications that have hooked in via element event controllers, can also request when ECNO shall update the calculation of interactions. In addition, the application can add and get event parameters.

Note that coordinations are directed and follows direction of the underlying class reference. Consequently, there is an important conceptual issue to understand in relation to interactions. In our example, it matters if we compute the interactions for CreateActivity by hooking into case or by hooking into control. While case does not have a coordination towards process, it could perform CreateActivity alone. Therefore, we have to hook into process when we designed it this way. Actually it would be possible to avoid this risk by modelling a bidirectional coordination, to say that none of the elements can do the event alone, but that is slightly more tricky than it sounds.

2.5.2 Summary and Advanced Concepts

We have just seen how ECNO allows us to model a behaviour of an application. ECNO defines two layers of behaviour, global behaviour and local behaviour. A global behaviour is expressed in a coordination diagram which looks very much like UML class diagram. This diagram type defines event types and associate them with classes (element types). In ECNO, classes are called element types, to make it clear we're not talking about ordinary classes. When an event type is defined in an element type, it means the element type may be able to participate in an event of that type. Notice, we said "may be able to" because it depends on local behaviour as well. For a graphical view of this, refer to Fig. 2.18.

Local behaviour can be expressed with ECNO nets, where transitions are extended with labels for attaching event bindings, condition bindings and action bindings as shown in Fig. 2.19. Note that the font type indicates the kind of the label. A condition binding is a boolean expression that restricts when the transition can fire (we didn't use it for CreateActivity, only for ExecuteActivity).

We also learned that events can take parameters, where parameter values can

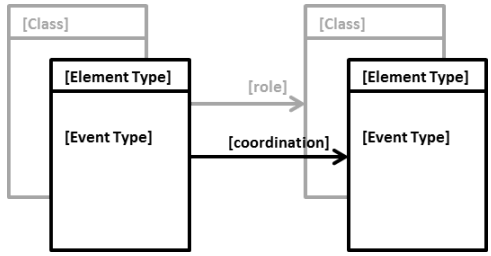

Figure 2.18: Notatation for global behavoir.

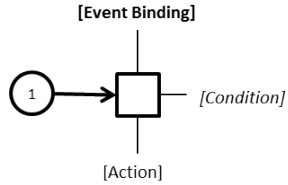

Figure 2.19: Notatation for local behavoir.

be injected by at most one (ignoring collective parameters) element and used by many other elements.

When all local and global conditions are met for an event, in a set of linked elements, we have an interaction, which can be executed by the application via an *element event controller*. In Chapter 5, we will explain the ECNO controller framework in much more depth.

In the example, we only used ONE-coordinations, which states that one linked element must to participate. There is another type, ALL-coordinations, which states that all linked element must participate.

At last, we learned that it is important to be aware of the trigger element, because coordinations per default are directed.

2.5.2.1 Advanced concepts

Synchronization: Events may be synchronised in the local behaviour of an element type. Syntactically, this is done in an event binding by including two (or more) event types, separated by semicolons, where the order is insignificant.

See the example below where parameters are omitted:

a = EventOne(); b = EventTwo(); ...

This prevents execution of any of them, until all listed events are enabled. And when they do execute, they do so together. Note that, only one event needs to be triggered and the others will be triggered automatically.

Element Inheritance: Element inheritance is modelled in coordination diagrams on element types, using same graphical syntax as for class inheritance in classical object oriented programming (an arrow with a hollow arrowhead). If *coffee* inherits from *brewer*, it will inherit the brewers global behaviour e.g. the ability to engage in certain events and coordinate with certain partners, but not its local behaviour. An element inheritance should always go along with a class inheritance for the underlying classes.

Event Inheritance: Event inheritance is one of the latest additions to the ECNO and is still ongoing work. The implementation, and also some concepts, have changed in the duration this master thesis. Especially, the mechanisms related to parameters. To explain the ideas we refer to Fig. 2.20, where we let Coffee inherit from Drink as an example. The effect is, that Coffee can re-use the behaviour of Drink, while adding some behaviour only used for coffee. How this plays out depends on how the local behaviour models of brewer and coffee elements refer to these two events. Since event inheritance is not applied in this thesis (was not available early enough), the knowledge is not strictly required follow this work, and we refer to the ECNO article [1] for further details.

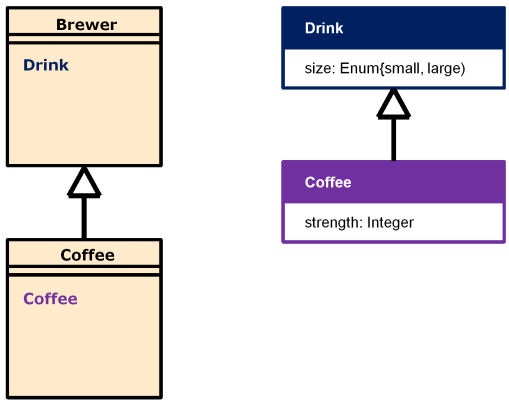

Figure 2.20: Element and event inheritance.

2.6 Related Work

In the previous sections the work that we build on was introduced. This section is dedicated to describe a part of the earlier work, which inspired the authors of ECNO and AMFIBIA.

2.6.1 Subject-Oriented Programming

The paper by Harrison and Ossher introducing subject-oriented programming [4], makes the claim that traditional object-oriented programming is insufficient for "the construction of large and growing suites of applications manipulating the objects". They argue that, in more complex systems, objects are simply too small to capture all functionality required in relation to the *identity* that the object represents. They go further to say that a client application using an object, may view this identity from a slightly different perspective than the author of the original class had anticipated. Quite often, it would not be an ideal solution to encapsulate new methods in the original class, especially if they are only relevant to one clients point of view. In turn, implementing the methods in the client application goes against the principles of encapsulation.

That is why the authors propose to "de-localise the concept of object" and instead apply perceptions - or *subjects* - when talking about an identity. The technology must allow subjective views and "emphasize more the binding concepts of identity to tie them together". Each subject can define its own data, behaviour and state to fit a point of view. However, subjects that are "composed in a universe" can *interact* by following composition rules. The authors discusses possible composition rules, but leaves it open to further work to formulate other implementations.

Subject-orientation uses object-orientation for realisation of subjects. It means that the behaviour of a subject is simply implemented by class methods. A subject *activation* "provides an executing instance of the subject". The article discusses that object creation and initialization within one subject, may also concern other subjects, and requires proper classification of the new object, in all subjects that use it (refer to [4] for details).

The ideas and concepts in AMFIBIA are very close to subject oriented programming (modelling) when AMFIBIA models aspects of core concepts independently and synchronize them. We could say that AMFIBIA applies the ideas of subject oriented programming while defining own composition rules when they syncronize the aspects. Even though AMFIBIA uses the term *aspect*

instead of *subject*, the relation to subject oriented programming is still relevant, because AMFIBIA cannot claim to be following the principles of aspect oriented programming. AMFIBIA's implementation techniques are different. Subject oriented programming is more open to interpretation and different implemetations of the ideas.

2.6.2 Aspect-Oriented Programming

Aspect-oriented programming was first introduced by Kiczales G., et al. [8]. We also use the book by Ian Sommerville [5] as a source on the topic. Aspect orientation is somewhat close to subject orientation at the conceptual level but it is more specific regarding implementation techniques. The starting point is also slightly different. Separation of concerns is acknowledged as good practise when designing software systems. It helps to isolate code that implement a feature to a smaller number of components. In the implementation phase, it becomes easier to split and delegate the work. In the maintenance phase an error is more easily tracked down to a few affected components. The problem is, that some concerns affects many of the systems components and cannot be dealt with in isolation. They are said to be *cross-cutting* concerns. In aspect-oriented programming, aspects are an abstraction that captures (or encapsulates) cross-cutting concerns and integrates them in the system.

As said, aspect-oriented programming is relatively specific about the implementation techniques and vocabulary. Very briefly, the code that implements a concern, and which is part of an aspect, is called an *advice*. In the executing program there are places (program lines) called *join points* where advices could be executed. Within an aspect, a *pointcut* then defines at which join points its advice shall be executed.

The ECNO article [1], relates itself to aspect oriented programming (modelling) by saying there is independence at the technical level, but "still, it was inspired by aspect orientation and is close in spirit to aspect oriented modelling or subject orientation". They argue too, that in a comparison ECNO events could be viewed as a kind of join points, while ECNO interactions are a kind of point cuts. However, they [1] emphasize that ECNO events are domain concepts (at the behavioural level), while join points are implementation artefacts added to the final program. They go further to say, that interactions are more symmetrical in nature than join points, since in ECNO, participating elements can add parameters or even block an interaction.

2.6.3 Process Algebra

The interactions in ECNO can be said to synchronize the processes of each participating element while also providing a communication channel for information exchange (event parameters). This is not a new idea at all. Communicating processes have been studied for many years and were formalised in process algebra. One of the best known contributions is presented in the book Communicating Sequential Processes (CSP) of C. A. R. Hoare (1985) [9]. The introduction to the concepts of processes contains the following quote: "Forget for a while about computers and computer programming, and think instead about objects in the world around us, which act and interact with us and with each other in accordance with some characteristic pattern of behaviour. Think of clocks and counters and telephones and board games and vending machines. To describe their patterns of behaviour, first decide what kinds of event or action will be of interest; and choose a different name for each kind." (P. 1). The relation to the central idea of specifying behaviour at a high level of abstraction, and more concretely to the events in ECNO, are quite obvious. CSP is a theory for mathematical modelling of processes in concurrent systems. In CSP, a process defines the behavioural pattern of an object, and is usually represented by a capitalized word/letter. An event is usually represented by an uncapitalised word/letter. The expression:

$$(x \rightarrow P)$$

says that, for an object, the event/action x leads to the process P. When x leads to the parallel execution of two processes P and Q, this writes as:

$$(x \rightarrow (P \parallel Q))$$

Processes communicates via *channels*. The following expression describes a process that outputs the message x on the channel c and then executes its behaviour:

$$(c!x \rightarrow P)$$

Here, $c!x$ is called a communication event. A process Q which receives x on c and uses the value of x in its behaviour can be written as:

$$(c?x \rightarrow Q(x))$$

The two processes will wait for each other at the respective statements and make the exchange on the channel (rendezvous). Leaving out x would just result in a synchronization without information exchange. CSP limits the number of processes, which can use the same channel to two: ("we shall observe the convention that channels are used for communication in only one direction and between only two processes." p. 114).

In contrast, ECNO coordinations work with any number of processes (defined in in elements), and there can be many consumers and contributors of the parameter values. The fact that ECNO models are defined on top of class diagrams, of which instances are allowed to update dynamically, gives us the freedom that the set of communicating processes (participants in ENCO) may change over time.

Chapter 3

Project Scoping

In this chapter, we detail the objectives of the workflow engine development project, and point out the limitations. First of all, we remind the reader, that we are not attempting to compete with the real workflow engines available today (see the argument in Sect. 1.2).

What is expected is an implementation which clearly can be recognized as being a workflow engine. This implies two modes: one mode for modelling of processes, a *modelling-environment*, and one mode for executing processes, an *enactment-environment*. Note that consequently, we have two different user levels. We shall call them *modelling users* and *enactment users*. Although we have the limitation of not integrating a real database, the implementation shall still emulate the presence of a database for storage of documents. We now define these components in the sections below. For each component, we discuss the objective and the limitations.

3.1 Modelling-environment

The modelling-environment is the component that allows the modelling user to define the business process models.

3.1.1 Objectives

It shall be possible to model the aspects of organisation, control and information by using an editor - that reflects AMFIBIA's lack of bias towards any aspects. The aspect models shall each have a feature richness that allows building somewhat realistic business process models. Traditionally, process loops is a challenge in BPM software, especially when they are data oriented. While we aim to implement a control aspect it shall be possible to overcome this challenge and support loops. While the current ECNO tool package is made for Eclipse, and while we can generate model editors from structural models using the Eclipse Modelling Framework (EMF), it will be a time saver to realize the modelling-environment EMF editor running within the Eclipse IDE, and so we chose this option from the beginning.

3.1.2 Limitations

Model validation may be omitted. Graphical model editor are may be omitted. A tree editor will be sufficient, since usability is not a main priority here. Extensibility and maintainability of our software is way more important. Also, conditional expressions written in textual form may be omitted, since this would require parsing of the expressions into structured form. Instead, in can be required of the modelling user to input the structured form directly.

3.2 Enactment-environment

The enactment-environment is the part of the system, which executes the business process models. It makes processes, cases, activities, and documents visible in a GUI for the enactment user. The GUI also reacts on enactment user input.

3.2.1 Objectives

Any number of processes, if any kind, can be executed in parallel. It shall be possible to run several instances of the enactment-environment at the same time (several users logged in at the same time). Not only the active instance, but also the other instances shall automatically update their content with minimal

delay [1]. It shall be possible to save the state of the enactment-environment on disk and restart it later. The enactment-environment shall have a presentable GUI that is able to demonstrate all features that the engine supports, such that any non-technical 3rd party would get an idea of what is happening. The GUI shall follow the WfMC *worklist* pattern. Different users shall be able log in and out and get their personal view of tasks. It shall be possible to open tasks and view and edit its documents.

3.2.2 Limitations

Invoking of external applications can be omitted. We shall not support distributed computing - e.g. every instance may run on the same machine. We will not support automated tasks, e.g. the user (an agent) is required to start and finish all activities.

3.3 Database integration (omitted)

In a real workflow engine, the database is where all data (incl. models) of the application is persisted.

3.3.1 Objectives

For the purpose of proving the information aspect, data shall be managed in a way that emulates the idea that documents are saved in a database. We also want to be able to re-launch the enactment environment with date saved earlier, which means, it shall be possible to save the state.

3.3.2 Limitations

To limit the scope in this thesis, we will not implement an interface to a conventional database. It will be left to future work to implement an interface to a relational database (or other type). The Hibernate [2] persistence framework for Java might be well suited for that purpose.

[1] We have no concrete performance requirements, since ECNO is not optimized for performance yet. The interesting part is which performance issues we face.

[2] http://www.hibernate.org

Since we are then only emulating a database, we have to relax the requirements to transactions, since getting that right without the help from existing technology would be beyond the scope (and purpose) of this thesis. Reliable database transactions must comply with the ACID properties:

A: Atomicity, C: Consistency, I: Isolation, D: Durability

It is beyond the scope to fulfil the durability property. That would for instance require committed data (here, the result of a task execution) to be stored even in the case of a power breakdown. That is not possible to guarantee, since we allow that saving the state requires some kind of user action.

The other properties are in principle compatible with the nature of ECNO interactions. The implementation of ECNO interactions (which could be seen as transactions here) does comply with the atomicity, consistency and isolation properties. But that does not necessary mean that execution of business processes inherit those properties - e.g. as will be show later, a task execution is not the same as a single interaction execution, if it was, then it would have been easier. This topic will be picked up again when we evaluate the workflow engine in Sect. 8.

Chapter 4

Workflow Engine

In this chapter we will present the conceptual contribution of this thesis. Based on the concrete process model given in Example 1, as well as on AMFIBIA, we will derive meta-models, to which we can add behaviour with ECNO.

Following the structure of AMFIBIA, our model for the core and each of the aspects are separated in different sub-models. The sub-models will be introduced in separate sub-sections, and we will mention the parallels and the differences to AMFIBIA in the text. We will as well separate formalism independent modelling and formalism dependant modelling.

For presenting the concepts of our workflow engine, we will use the same notation as in the ECNO introduction in the Chapter 2. This notation is an ad-hoc diagram type combining structural models and global behaviour models. In the real implementation (in Eclipse/ECNO modelling tools), these diagrams are separate e.g. ECNO diagrams are referencing conventional EMF Ecore diagrams. But, our notation here is more compact and therefore useful for presentation purposes. In Sect. 6.4.1 we show how the real models look.

4.1 Patterns

This section explains the main patters applied when modelling. The purpose is to prepare the reader for the following sections where the patterns are used, by giving a high level introduction here.

4.1.1 The "instance-of" stereotype

In Sect. 2.4.2.1, we introduced AMFIBIA's meta-models and models for runtime information. They made use of a relation named "instance-of". Since a full understanding of this concept is important for seeing the broader picture in our models we will elaborate on it here. Please refer to Fig. 4.1. The figure will be explaied in the following paragraphs.

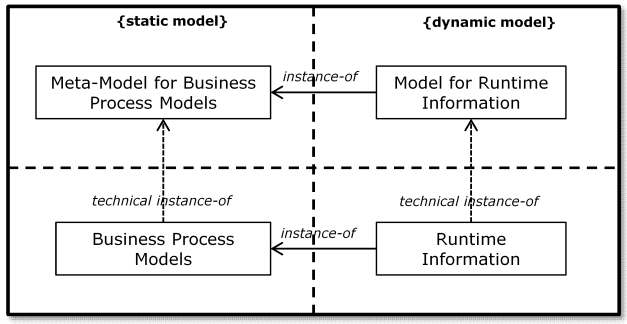

Figure 4.1: The "instance-of" relation.

Recall Example 1 (Sect. 2.3) defining a business process model for managing errors. If that business process model were being executed in an enactment environment, that would produce runtime data - or runtime information. The idea above is captured in AMFIBIA's meta-model. Recall from Sect. 2.4.2.1 that AMFIBIA defines a meta-model for business process models, but in addition to that, it defines a model for the runtime information that may be created for it. AMFIBIA use the "instance-of" stereotype to express that a concept in the runtime information model conceptually is an instance of (has the type) of a concept in the business process meta-model.

The "instance-of" relation is only conceptual and should not be mistaken for the technical instantiations which also occurs at some point, where classes are instantiated into objects. In this respect, the objects of the business process mod-

els are technical instantiations of the classes in the business process meta-model. Likewise, the objects of the runtime information created by the enactment environment are technical instantiations of the classes in the runtime information model. Consequently, the objects of the runtime information are conceptually instances of the objects of the business process models.

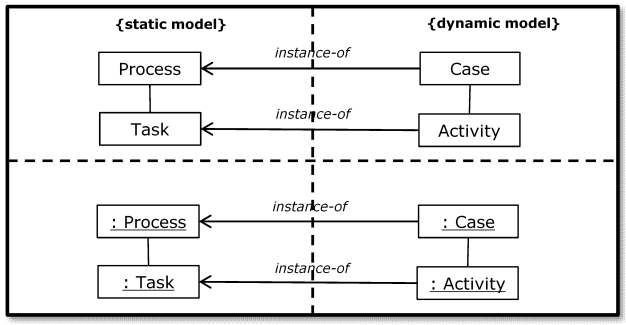

Figure 4.2: The "instance-of" relation with concrete BPM concepts added.

In Fig. 4.2 we have replaced the names of the content of each of the four quadrants in Fig. 4.1 with some concrete classes and objects of our domain. As can be seen, the business process meta-model includes the concepts of process and task while the runtime information model includes the concepts of case and activity. The business process models includes technical instantiations of process and task, while the runtime information includes technical instantiations of case and activity. Note that each of the four concepts mentioned have aspects too but they are omitted in the figure.

This chapter is concerned with modelling the concepts of business process models. These concepts are belonging in the upper half of the figures. Terminology wise, we may often use shorter names, referring to the left side as the static model, and referring to the right side as the dynamic model. The pattern of only referring in the direction from the runtime information model (dynamic model) to the business process meta-model (static model), will be followed in all our models, because it is consistent with AMFIBIA (actually, it would be conceptually wrong if types referred to their instances). A vertical dotted line will be used to separate the two sides.

4.1.2 The "aspect-of" stereotype

Here we explain the *aspect synchronization* pattern we use throughout our models to realize the "aspect-of" stereotype presented in AMFIBIA. Recall from Example 1 (Sect. 2.3), that aspect oriented modelling made it possible to model the same business process from different points of view. We made separate models for the organisation, the control and the information aspects. Of course, a system which executes according to such models shall be able to synchronise the behaviour appropriately, and AMFIBIA proposed concepts for this. Refer back to Sect. 2.4.3 to see how ABFIBIA used a composition for the structural component of their "aspect-of" relation and synchronized their aspects using an automata based technique. With ECNO, the latter part can be achieved in a much simpler way with ECNO events and ALL-coordinations, as shown in Fig. 4.3.

Figure 4.3: The "aspect-of" relation with ECNO based synchronisation.

In the figure, *CoreElement* is representing process, task, case or activity while *ElementAspect* is representing aspects of any of these four core concepts. The general pattern is, that we define the same event type in the elements that should be synchronised, and model with an ALL-coordination, that if the core element is doing this event, then all its linked aspect elements must participate. Put in other words, we achieve synchronisation with respect to an event type among coordinated elements. When ECNO finds an interaction formed by elements that can participate, the elements will perform their actions together if that interaction is executed. Later, in Fig. 4.7, we will see a concrete use of this technique.

4.2 Architectural overview

The after this section we will go into details with the models realizing the workflow engine. However, it makes sense to first give a high level overview of how these models are going to work in the bigger picture. We've already placed our main concepts in four different quadrants as shown in Fig. 4.1.

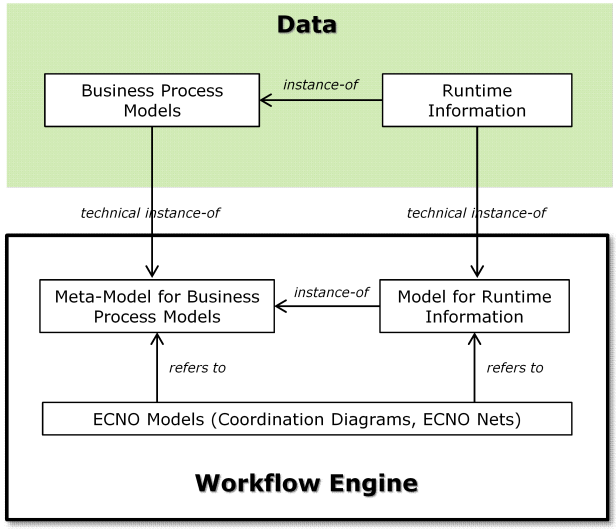

Figure 4.4: Architectural overview.

In Fig. 4.4 we have grouped the meta-model for business processes, the runtime information model and the ECNO models inside the component *workflow engine*. We have done this to make it explicit, that they realize the workflow engine. In addition, we indicate in the figure, that the business process models and the runtime information contains the data that our workflow engine uses.

4.3 Realizing the Core

In this section, we will present the models of the core. It might be worthwhile to refer back to AMFIBIA's core model in Fig. 2.5 and aspect integration model in Fig. 2.6. It will then be more obvious, that our structure is conceptually close. Recall from the section about AMFIBIA, that the overall purpose of the core is to single out the concepts that are common to all aspects, and to be an anchor point for the integration of aspects. In our presentation of the core models, we will first present the structure of the static and dynamic part of the model, which is practically a copy of AMFIBIA's core model. Then we will present the global behaviour of the core, and finish with the local behaviour of the core.

4.3.1 Structure of the static model

In Fig. 4.5, the static core model captures the concepts of *process* and *task*, where tasks are contained in processes. The model additionally captures the the integration of *process aspects* and *task aspects*. We use compositions to model the "aspect-of" relations. Process aspect and task aspect are interfaces, and they are implemented by concrete aspect elements in concrete aspect models. This construct is flexible since the system can always be extended with additional aspect models (recall, there are more that three) by implementing process aspect and task aspect in an extending model.

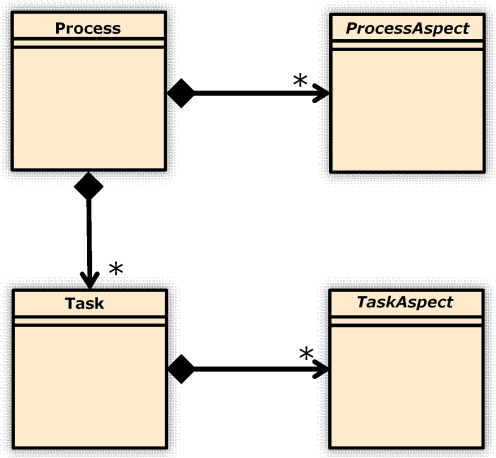

Figure 4.5: Core (static).

In Fig. 4.6, the interface process aspect is implemented by *control process aspect*, *information process aspect* and *organisation process aspect*. Notice the convention of adding the capital letter "I", "O" or "C" after the element name, to indicate which kind of aspect package it belongs to. For tasks aspects the realisations are *control task aspect*, *information task aspect* and *organisation task aspect*. We have omitted that figure while it identical to Fig. 4.6 except the word "process" shall be replaced with "task".

We will use the term *identity* to address a core element and its aspect elements combined. Specifically, instances of Task (core) and its referred TaskO (organisation), TaskC (control) and TaskI (information) instances together are referred to as a task identity. The four elements are talking about the same tasks, just from different points of view.

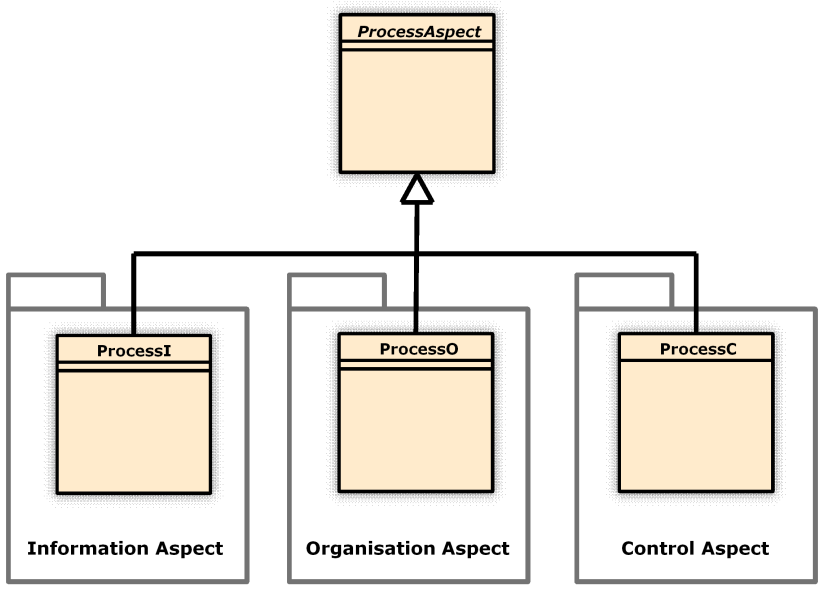

Figure 4.6: Process aspect realizations. Task aspect, case aspect and activity aspect have the same construct.

4.3.2 Structure of the dynamic model

The dynamic model expresses a pattern, which symmetrical to that of the static model, as is our comments. In Fig. 4.7 (ignore the behaviour for now), the dynamic core model captures the concepts of *case* and *activity*, where activities are contained in cases. The model additionally captures the the integration of *case aspects* and *activity aspects*. Again, we use compositions to model the "aspect-of" relations. Case aspect and activity aspect are interfaces, and they are implemented by concrete aspect elements in concrete aspect models. Again, this construct is flexible since the system can always be extended with additional aspect models by implementing process aspect and task aspect in an extending model. In addition, case and activity have "instance-of" relations to their conceptual types process and task. Refer again, if needed, to the design pattern discussion earlier in this chapter.

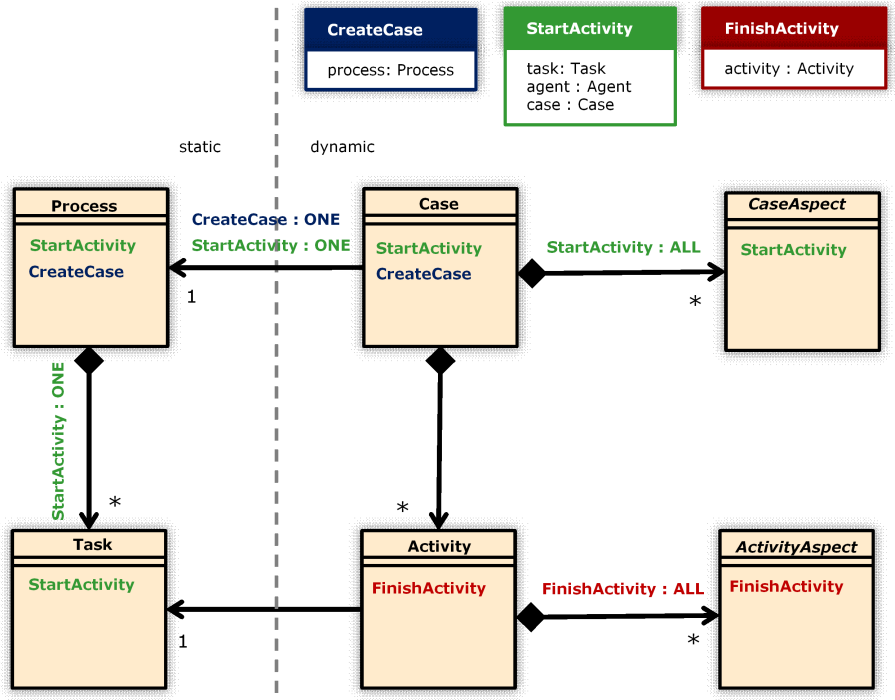

Figure 4.7: Core (dynamic).

In our implementation, case aspect is implemented by *control case aspect*, *information case aspect* and *organisation case aspect*. Activity aspect is implemented

by *control activity aspect*, *information activity aspect* and *organisation activity aspect*. The constructs are similar to the figure showing the implementations of process aspects - Fig. 4.6 - except the word "process" shall be replaced with "case" and "activity" respectively.

We will now come to the more interesting part, which is that of defining behaviour in the core.

4.3.3 Global behaviour

This section will present the global behaviour of the core. We will start with an analysis before we present the models. Actually, the behaviour of the core is the most important to get right, because it affects all the other models (the aspects).

4.3.3.1 Analysis

ECNO does not come with a methodology for behaviour modelling (although there are plans of making one), but in order to get started with the modelling anyway, we could remember the words of C. A. R. Hoare in his introduction to CSP [9]: "first decide what kinds of event or action will be of interest; and choose a different name for each kind." This advice sounds like it could apply here, and recall the similarities between ECNO and CSP mentioned in Sect. 2.6.3.

Fortunately, we know something about what kind of events or actions takes place in a workflow engine, for instance, from AMFIBIA and from WfMC. We know that the core functionality of a workflow engine is to execute cases based on process models, and execute activities based on tasks in the models.

Still, we cannot define ECNO events from the behaviours "execute case" and "execute activity". The reason is, that ECNO interactions must execute instantaneously (see discussion in 4.7.4). A case execution takes a certain amount of time, and so does an activity execution. However, we could break down a case execution and an activity execution into smaller pieces, which could be treated as zero-duration events.

We break down the behaviour in "execute activity" into events named "start activity" and "finish activity". For reasons that will be more clear later, we capture the behaviour of "execute case" with the event named "create case", plus

some behaviour that we do not capture explicitly with events. In the models, and in the rest of the text, we will use the notation StartActivity, FinishActivity and CreateCase to refer to our event types.

Until this point, we have derived which event types we could use in the core. However, we have not explained in which elements types they should triggered, or with which other element types this trigger element type should coordinate (refer to ECNO concepts in 2.5.2 if needed). We will come to that in the following section, when we comment on the models.

Before we can do that, we have another concern to consider up front. In particular, that of when to instantiate [1] case and activity respectively. It turns out, that the instantiation concern is not very trivial to explain. Therefore, we will state our strategy for instantiation below, and discuss the arguments in Sect. 4.7.7 and Sect. 4.7.6, because the discussion would be a sidetrack right now.

We choose that activities are instantiated (or, created) in the same moment as they are started by the user - e.g they do not exist before they are started. We capture both aspects, creating them and starting them, in the event StartActivity.

With respect to cases we define two states, *initiated* and *active*, we will see this more clearly in the ECNO nets later. We choose that cases are instantiated automatically, such that every process instance (business process model) known to the system has exactly one case instance in initiated state. Cases are then transitioning to active state the first time an activity is started in them. At this point we will instantiate a new case, to maintain the invariant that every process instance has a corresponding case instance in initiated state. We capture the creation of cases with CreateCase.

4.3.3.2 Model

In Fig. 4.7 we had already added the global behaviour of the core and ignored it at first, but here we can explain it. Note that the colours we use in the diagram, are only for readability purposes. We will explain separately for each of the three event types defined in the analysis.

First, however, we will say that StartActivity is triggered in case and FinishAc-

[1] The reader might ask if we mean technical or conceptual instantiation. Actually both. The new elements in the runtime information are technical instances of element types in the runtime information model, and conceptual instances of elements in the business process model.

tivity in activity. This choice is partly explained by the instantiation concern mentioned above. At least, that explains why StartAcitivity could not be triggered in activity - e.g. the activity does not exist until StartActivity has executed. We discuss alternatives for the choice of the trigger elements in the core in Sect. 4.7.2.

So, StartActivity is triggered in case (by the GUI), and also represented in case aspect, process and task. Case has an ALL-coordination for StartActivity towards case aspect. Recall, or refresh, from Sect. 4.1.2 that we use this pattern to synchronize a core element with its element aspects. Further, case has a ONE-coordination for StartActivity towards process, which again has a ONE-coordination for StartActivity towards task. Here, we only really need the participation of task, but we use process to navigate to task. We will explain in the next section, why task is needed here. Finally, StartActivity takes three parameters: a task, an agent and a case.

FinishActivity is triggered in activity (by the GUI), and also represented in activity aspect. Activity has an ALL-coordination for FinishActivity towards activity aspect. FinishActivity takes one parameter: an activity.

CreteCase is triggered in case (by synchronisation with StartActivity), and also represented in process. Case has an ALL-coordination for CreateCase towards process. CreateCase takes one parameter: a process.

4.3.4 Local behaviour

We here continue by defining the local behaviour of each of the element types, starting with a short analysis of the life cycles of our core elements.

4.3.4.1 Analysis

With regards to the life cycles we are inspired by the WfMC Reference Model (see [10] Fig. 7 and 8), but our models are simplified for our purpose.

Recall that when an activity is created, it is also started. Therefore, we can say that an activity can be only *active* or *completed* - in that order. The transition from active to completed occurs when it participates in FinishActivity. A case has the states *initiated* or *active*. The transition from initiated to active occurs when it, for the first time, participates in StartActivity. Recall that we defined an active case, as one having at least one started activity. In our design, process

and task only have one state each, but we will anyway use ECNO nets to characterise them.

4.3.4.2 Models

We now model the life cycles of the core elements with ECNO nets (refer to 2.5.1.6). Recall from Sect. 2.5.2 that ECNO nets makes it easy to create an explicit life cycle model, and to bind events, conditions and actions to the (Petri net)-transitions in the ECNO net.

In Fig. 4.8, we model that process can locally always do StartActivity. Actually, this is just used to accomplish that we can navigate to task via process when looking for interactions. Process can do CreateCase when the supplied parameter is equal to itself, and the bound action creates the case by calling a code snippet.

In Fig. 4.9, cases are created in the state *initiated* in which they can participate in StartActivity synchronized with CreateCase. The case itself is injected into the case parameter for use in process (note: it might seam redundant to inform process about the case, but it allows for another use case - that of creating selected cases from the GUI). Recall, we wanted to create a new case of the same type when the first activity starts. When cases have transitioned to active state, they can always do StartActivity, but without starting new cases.

In Fig. 4.10, task can always do StartActivity and it will inject itself to the parameter task. The reason for injecting task here, is that ultimately we only want to ECNO to compute interactions, which each involves a single task element, we discuss this in more depth in Sect. 4.7.1.

In Fig. 4.11, activities are first active state, and they can one time participate in FinishActivity requiring that the supplied activity parameter is equal to itself. This pattern expects the GUI to inject the activity which the user wants to finish [2]. The action binding just sets a local variable.

4.3.4.3 Evaluation

Notice in the action bindings of transitions that binds to StartActivity we're calling a code snippet *createAcivity(..)*. This method creates a new activity

[2] Note that, when we trigger FinishActivity in activity from the GUI, the equality check, and hence the activity parameter, is a bit redundant, and it could be taken out of the design.

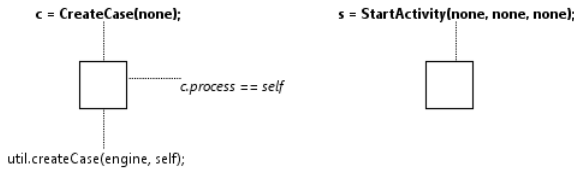

Figure 4.8: Local behaviour of process.

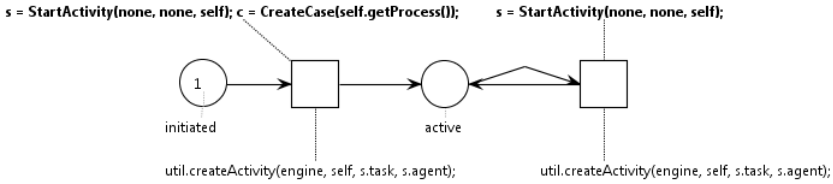

Figure 4.9: Local behaviour of case.

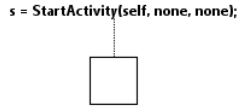

Figure 4.10: Local behaviour of task.

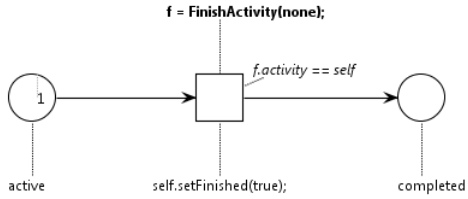

Figure 4.11: Local behaviour of activity.

element in the core, but we're also forced to create activity elements on behalf of the aspects. This violates the separation of concerns principle. ECNO does not include a concept that allow elements that participates in an interaction to cooperatively build a data structure when that structure includes new elements. Ideally, to separate concerns, the case element of the information aspect (a participant of the StartActivity interaction) would have created the activity element which belongs to the information aspect and attached it to the activity element of the core. However, we could not find a suitable way to make the (information)-case aware of the newly created (core)-activity. Note that we did not want to add the new activity to the parameter list for efficiency reasons - e.g. it would be created every time the interactions are evaluated.

4.4 Realizing the Control Aspect

We will now continue by introducing one of the three aspect models included in this thesis. For the control aspect, we first present a model which is independent of specific formalisms which could be used for modelling the control aspect. This is followed by a Petri net implementation inspired by AMFIBIA and the article: An ECNO semantics for Petri nets [11]. Note that, in this section we do not consider our use of ECNO concepts to be a formalism for modelling control. Like class diagrams, ECNO diagrams could be used to define a multiple of formalisms.

4.4.1 A formalism independent model

This section is based on the formalism independent control model in AMFIBIA, and basically adds ECNO concepts on top of it. As before, we start with a short analysis, before presenting this model.

4.4.1.1 Analysis

First of all, we want to include a formalism independent model in this thesis because we want to make it easier to integrate other formalisms for the control process than the one we have selected. Therefore we have to capture the concepts that are common to all formalisms.

A central concept of the control aspect is that of activated tasks. An activated task is one that may be started in a given state. AMFIBIA maintains an ex-

plicit reference from state to the activated tasks (see Fig. 2.7). Furthermore, in the control model, AMFIBIA tasks have operations *initialized* and *finalize* taking a state as parameter and returning a new state. This implies that the reference to the activated tasks are updated in the returned state according to a control process model. In this section we aim to re-capture the basic elements of the control aspect, and transfer above mentioned mechanism into ECNO like behaviour.

4.4.1.2 Model

The model for a formalism independent control aspect is shown in Fig. 4.12. Relative to the previously seen models, the only new element is that of *state* contained in case, and this is not a surprise because we just follow AMFIBIA there. A local event type StartActivityC is added and synchronised with StartActivity in (control)-case. The reason is that we need an event type taking also state as a parameter, but it should be known only in the control aspect to maintain separation of concerns. Similarly, FinishActivityC is synchronised with FinishActivity in activity (control aspect).

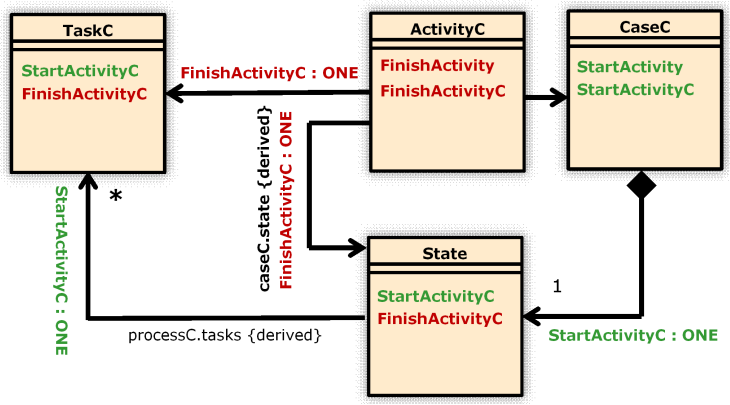

Figure 4.12: Control aspect - formalism independent model.

Regarding the notion of activated tasks, our model varies slightly from AMFIBIA. In our model state has a reference to the process which has a reference to all tasks, not only the active ones like in AMFIBIA (in the figure a derived reference directly to task). Since we now use ECNO, the activated tasks are instead given by the possible StartActivity interactions in a given state. As in AMFIBIA, it is the responsibly of task to calculate if it is activated in a

given state, therefore the element state injects itself into the parameter named state, to make sure state is available in the task. In the formalism independent model we don't specify how task uses state to conclude on this. It might be a local behaviour, a local operation or it might involve coordinations with other elements. When StartActivityC is executed, we expect the task to change the state accordingly (like AMFIBIA's initialize operation in task). FinishActivityC follows an identical pattern (and like AMFIBIA's finalize operation in task), except activities can always finish. Still, state is put in the parameter since task is expected to change the state when FinishActivityC executes.

4.4.2 An implementation

The objective of this section is to demonstrate how the formalism independent model can be extended with an actual ECNO based implementation from which we can generate executable code. For this thesis, Petri nets were selected as the formalism of the implementation of the control aspect. We selected Petri nets because they are based on simple yet powerful concepts for modelling flow and because they support concurrent execution.

4.4.2.1 Analysis

We want to integrate the concepts in the article by Kindler [11] into the design of our workflow engine. Basically, the article takes the Petri nets concepts of places, arcs and transitions, as well as their relations, and defines the events Fire, Add and Remove on top of them in a coordination diagram (global behaviour). The event Fire is referring to the transition firing concept of Petri nets, while Add and Remove are referring to the actions of removing tokens at the input places and adding tokens at the output places. In transition, Fire is then synchronised with Add and Remove, since the firing of a transition is exactly the combination of these two actions.

In order to integrate these concepts into the workflow engine, more specifically into the control aspect, we have to define how the Petri net concepts maps to the BPM concepts. Actually, this topic is well covered in the literature, for instance Workflow Nets that are discussed in the book of Weske [12]. AMFIBIA discusses the topic as well. What remains, is the work of combining the existing ideas into a working executable ECNO based model for a workflow engine.

In the book of Weske [12] as well as in AMFIBIA, tasks (activities) are mapped to Petri net transitions. We will follow this principle. Weske discusses the issue,

that we also have, of the build-in contradiction in representing tasks with Petri net transitions. In particular that the execution of tasks consume time while the firing of a transition per definition does not. In other words, if we allowed the invocation (starting) of a task to map to a transition firing, it would have the consequence the the following tasks would enable to soon. They should not be enabled until the invoked tasks finishes. In our case, we can solve this problem in a elegant way as will be show in Sect. 4.4.2.3.

4.4.2.2 Model: Structure

Please refer to Fig. 4.13 which shows the model. In the model, we let *petrinet e.g. Petri net* implement the control aspect of process, and let *transition* implement the control aspect of task. Finally we let *marking* implement state.

We introduce the *arcs* and *places* contained in the Petri net and model that transitions can have a multiple of input and output arcs, whereas arcs can have one source or one target place. The marking contains a list of *tokens* which have each a reference to the place they are located in.

Recall that the static model cannot refer to the dynamic model according to the pattern we use. This is why we have to refer from token to place, and not the other way which is more common.

4.4.2.3 Model: Global behaviour

Now we move on to the global behaviour. In the analysis above we mentioned the error it would be to map StartActivity to transition firing. To avoid this problem we choose to ignore that Petri net transitions normally fire instantaneously. We simply map StartActivity to Remove and we map FinishActivity to Add. It has the nice effect that the following tasks do not enable until a tasks has been finished.

We assign the events Add and Remove to transition, arc and place. These events are the two components of the firing behaviour of Petri nets. Remove refers to the consumption of exactly one token at every source place of a transition. Add refers to the production of exactly one token at every target place. This behaviour is valid when Add and Remove are triggered in transition. In the case of Remove, transition requires the participation of ALL ingoing arcs which in turn requires the participation of their source place (which need to be populated with at least one token, see local behaviour). In the case of Add,

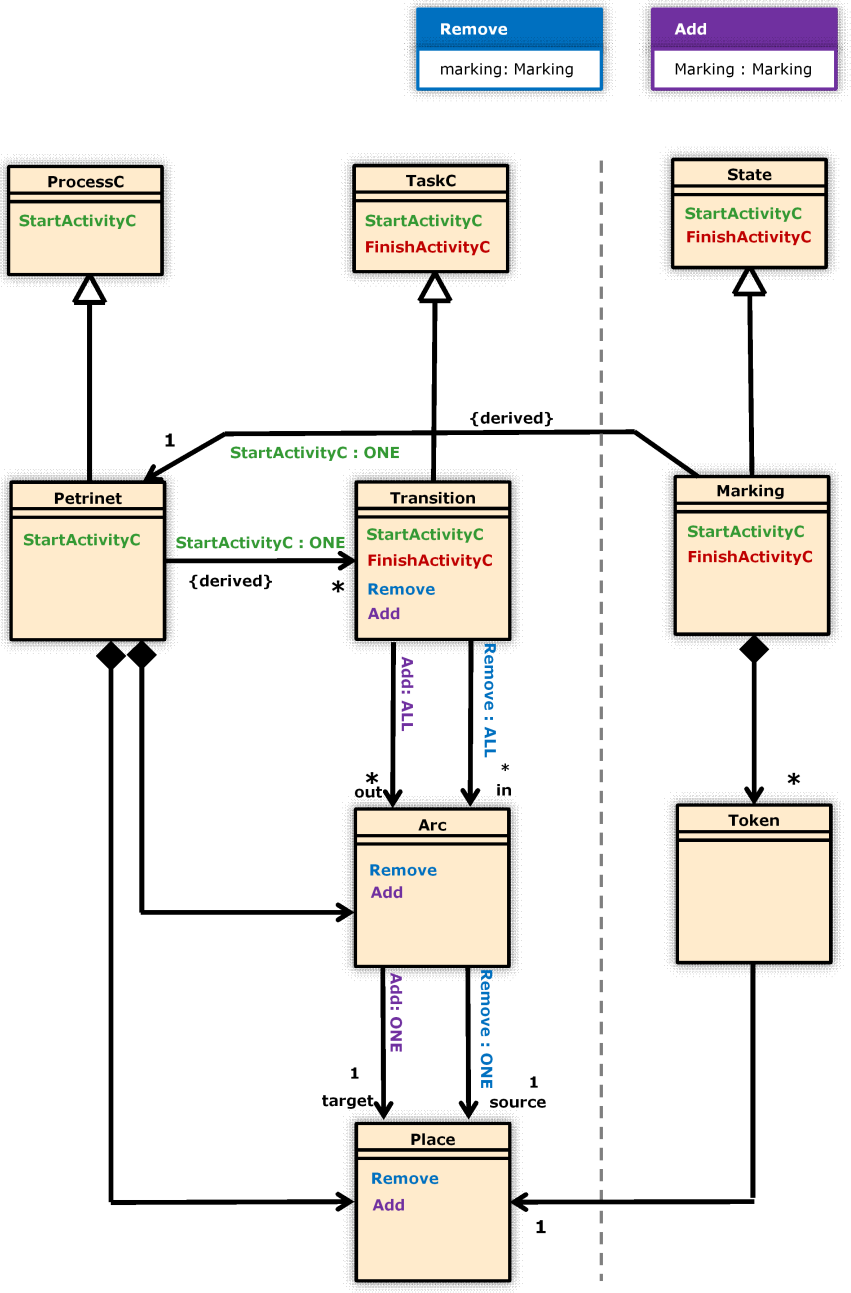

Figure 4.13: Control aspect - a Petri net implementation.

transition requires the participation of ALL outgoing arcs which in turn requires the participation of their target place.

4.4.2.4 Model: Local behaviour

The local behaviour of marking, transition and place can be seen in the figures 4.14, 4.15 and 4.16. Local behaviour in petrinet and arc just defines unconditional participation and are therefore omitted.

In marking, StartActivity and StartActivityC are synchronized, while marking injects itself as parameter in StartActivityC - this follows the principle given in the formalism independent model. In addition the task parameter is transferred to StartActivityC.

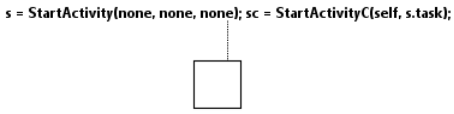

Figure 4.14: Local behaviour of marking.

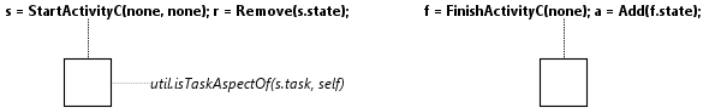

Figure 4.15: Local behaviour of transition.

Figure 4.16: Local behaviour of place.

In transition, StartActivityC is synchronized with Remove and FinishActivity with Add, while passing on the state (marking) parameter. With this, our Petri net model now governs when tasks can start and finish. Notice also that in

transition there is condition testing that the transition is a task aspect of the task in the event parameter. Recall that in the core model, a task were required to participate in StartActivity. Here we're checking that the control task (the transition) belongs to the same task identity. Had we left out this check, we would have allowed multiple task identities in the same interaction which would not be correct.

In place, the Remove event can only enable the associated condition evaluates to true. The condition uses state (marking) element in a code snippet which checks that the place contains a token. The action takes care of removing the token when an interaction including this place executes. The event binding to Add has an associated action which adds token to this place. This can occur unconditional because a place may legally contain more that a single token.

4.4.2.5 Model: Summary

The presentation of the models included a lot of details, so we will end this section with an overall summary. Actually the control model is quite simple: It includes only known Petri net concepts (arc, place, transition, marking, token). They were connected structurally following known practise for Petri net meta-models, except for token that we could not refer to in the static model. ECNO coordinations were used to express adding and removal of tokens when transitions fire. From transition to arc we used ALL coordinations on Remove and Add. From arc to place we used ONE coordinations on Remove and Add. Since we could not refer to token directly in the coordination diagram, we passed the marking into Add and Remove as a parameter instead. We hooked the Petri net into the workflow engine by implementing PrecessC, TaskC and State of the formalism independent model, and by synchronising StartActivity with Remove and FinishActivity with Add.

4.5 Realizing the Information Aspect

In the information aspect we again start with a formalism independent model and present an implementation thereafter.

4.5.1 A formalism independent model

We divide this section into an analysis and a model presentation respectively.

4.5.1.1 Analysis

AMFIBIA defines a model (Fig. 2.9) that we will build on here. Refer to Sect. 2.4.2.4 where we explain the AMFIBIA concepts for documents and in particular *document descriptors*. Or, recall that document descriptors returns documents of a task given a context.

As in the information aspect we mainly want to adapt the AMFIBIA model and add support for ECNO behavioural modelling. For the formalism independent model in particular, this mainly reduces to how we can link our StartActivity and FinishActivity to document descriptors. Recall from the discussion in Sect. 2.3, that input documents of a task are required to start that task, while output documents are required to finish a task. Therefore is makes sense to connect our StartActivity to the existence of input documents and FinishActivity to the existence of output documents.

AMFIBIA's document descriptors have an operation that returns the document, and the documents are being added (by the information aspect) to the corresponding (information)-activity when the tasks instantiates into an activity. Ideally our information aspect should also have been responsible for attaching the found documents to an (information)-activity. However, recall from the discussion in Sect. 4.3.4.3, that in our implementation the core creates the (information)-activity during an execution of a StartActivity interaction.

This is why we will up-front allow ourselves to ignore the issue of separation of concerns in this respect, and likewise attach the documents from within the case in the core. Ideally the core should not have been aware of documents.

Finally, we want to ensure that document and document types are global concepts e.g. not owned by a single process.

4.5.1.2 Model

In Fig. 4.17 we start by defining global containments for *document types* and *documents*, before we use them in the process model. Document types are placed in the container *templates* and documents in the container *data*. Actually, with this trick, we're just emulating the use of a database, which is also a global storage place. We leave it up to future work to implement an interface to a real database.

In Fig. 4.18, the elements task, activity and case are present in their informa-

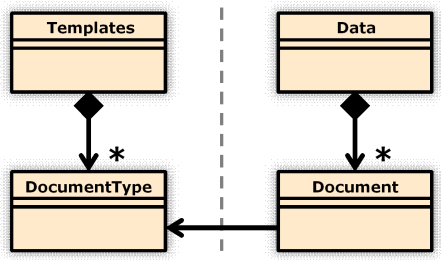

Figure 4.17: Information aspect: Global concepts.

tion aspect versions. Task contains input- and output document descriptors, for which realisations must "use" (refer to) document types. In the runtime model, activities refers to their input and output documents, which refers to their individual document type. In case StartActivityI is synchronized with StartActivity and takes two parameters case and task. StartActivityI in case is coordinated with StartActivityI in task. To perform this event, task requires the participation of all input document descriptors. From activity, a symmetrical pattern is modelled for FinishActivity concerning the output document descriptors.

As can be seen, the event StartActivityI takes two parameters, task and case. If an input document does not exist, a document descriptor shall block the interaction (refuse to participate). While if an output document does not exist, the interaction shall be valid because the user should have the option to create them from within the task (refer to the discussion in the example). Exactly how the GUI is made aware of non-existing output documents in our implementation will be explained in the following section.

Note that we have omitted complex documents, and document relations known from AMFIBIA for the sake of simplification.

4.5.1.3 Evaluation

We did not achieve to copy the principle in AMFIBIA where document descriptors returns documents. Normal ECNO parameters does not allow the multiple document descriptors to contribute with a parameter value.

However, we could (and probably should) have used a feature in ECNO called *collective parameters* which allows many interaction participants to contribute to the same parameter, where the result would be an unordered list of values.

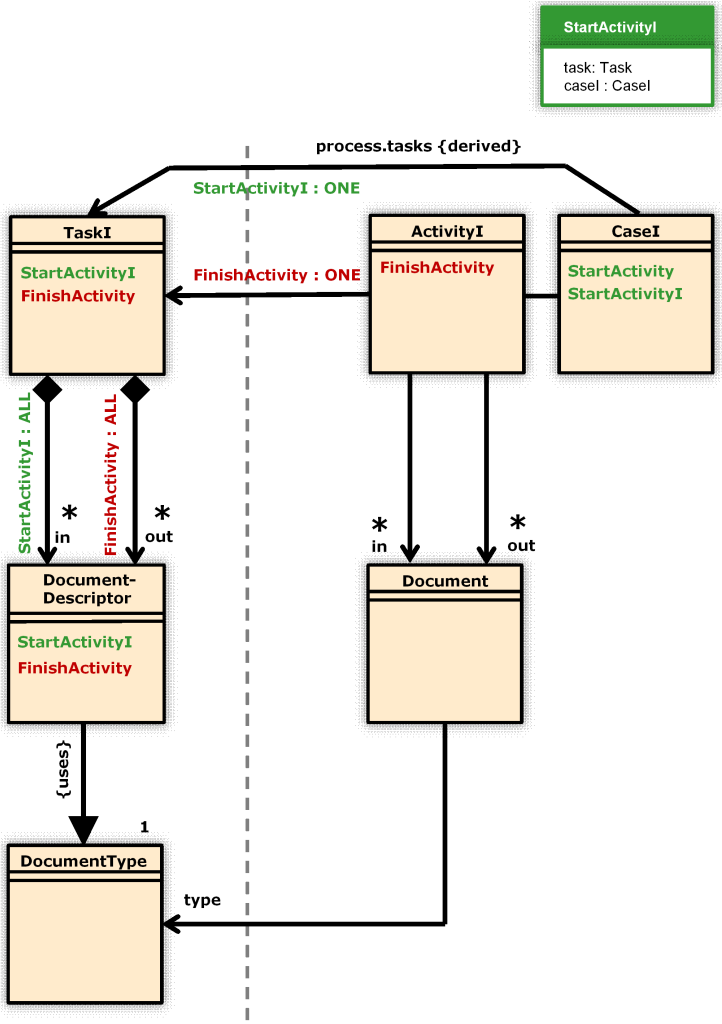

Figure 4.18: Information aspect: formalism independent model.

An implementation, which use our model above, need to iterate through the document descriptors again when the interaction executes, which is a slightly redundant compared to just having the documents available in a collective parameter. Note that, here we are taking about documents that already exists, so we would not have the efficiency problem mentioned in the analysis - e.g contributing a newly created activity to a parameter. We will leave it open to future work to implement a parameter for documents.

4.5.2 An implementation

In the follow subsection we present our implementation of the information aspect. The implementation is divided into the concerns of implementing a *process document descriptor*, a *document type*, and a feature for adding *document condition*. We will limit this implementation to documents within a process - e.g. *process documents*.

4.5.2.1 A process document descriptor: Analysis

Recall that we have earlier seen the concept of process document in Example 1 where, for instance, *error_report* was a process document. This is why the simple document descriptor - which refers only to process document - is sufficient for creating a meta-model supporting Example 1, where all documents are defined within the same process.

A realistic information model might also refer to documents of another process, or documents that are not even part of a process, but just exists in the database anyway. We have aimed that our meta-model can be extended with document descriptors of this kind but we leave that to future work. It could be done by extending or replacing the formalism for document descriptors that we introduce here.

4.5.2.2 A process document descriptor: Model

Fig. 4.19 shows an example realisation of a document descriptor pattern. The process document descriptor refers to a process document that is contained in process. Process document documents are characterized by a qualified (in the process) name and by a document type. We have also introduced a qualified

reference from case to document, which refers by name to process documents that have been created in or loaded into a case at a given time.

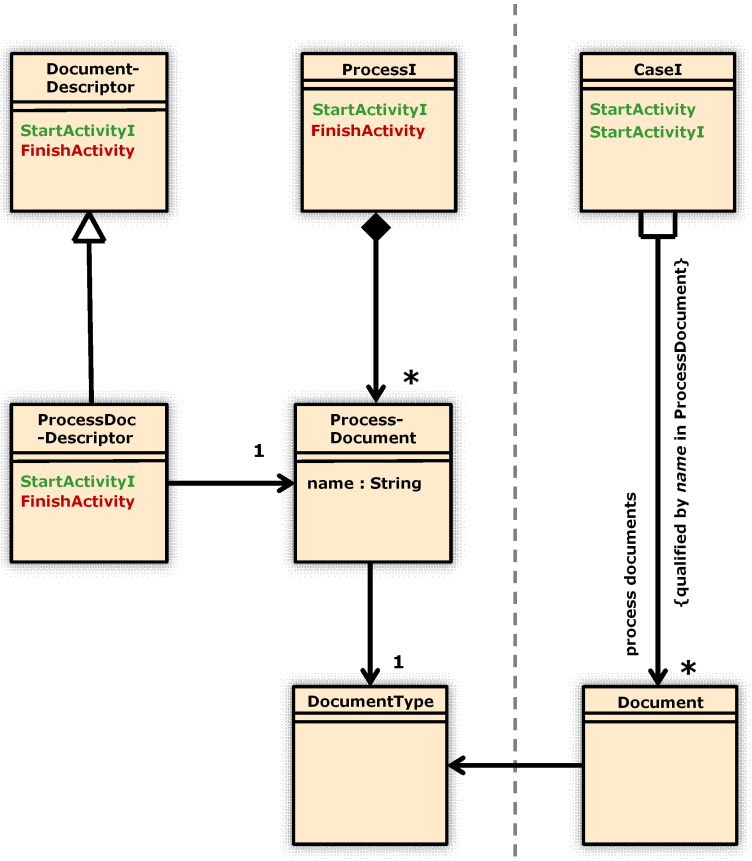

Figure 4.19: Information aspect: implementation of a process document descriptor.

In Fig. 4.20 the local behaviour of a process document descriptor implements conditions for StartActivityI and FinishActivity that in both cases calls local method *getDocument(..)*. The method is very simple, it just performs a lookup in the qualified map of documents attached to a case and returns the document, if it can be found. As key in this lookup the method uses the *name* attribute in the process document descriptor. Notice that StartActivity is only implemented in input document descriptors, and FinishActivity only in output document descriptors!

Note also that, the GUI (Task Viewer) can create or import the non-existing output documents and attach them.

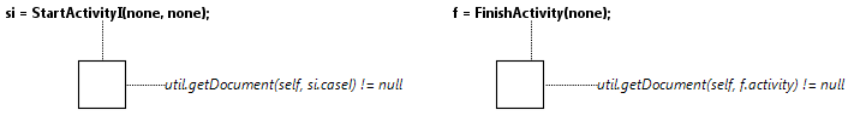

Figure 4.20: Local behaviour of a process document descriptor.

4.5.2.3 Document types

Here we will present a simple implementation of document types which comes with this workflow engine. Note that we have seen document types before in Example 1 where for instance ErrorReportDoc was a document type for the process document *error_report*. Alternative terms for document types, could be forms or schemas for information.

The implementation of document types can be seen in Fig. 4.21.We call this specific implementation a default document type. It is limited to content - e.g it is not possible to model layout. The model for default document type basically consists of fields and enumerated fields. Enumerated fields consists of literals and a reference to the default value. The instance model consists of classes to hold the values for each of the mentioned modelling classes. Refer to the image to see the structure.

4.5.2.4 Conditions: Analysis

Recall from the example that we sometimes want the express in the information model that additional conditions (besides the existence of a document) must be met before a task can start or finish. For instance, refer back to Example 1, where the task Clarify cannot not start unless the field *decision* in the document *error_report* is equal to "clf". We could express this condition as:

Start Condition: error_report.decision == "clf"

We have chosen to separate the concept of conditions from the concept of document descriptors. The main argument is, that a task could have start or finish conditions, which depends on values in more than one document. In such cases there would be the question of which document descriptor to express this in.

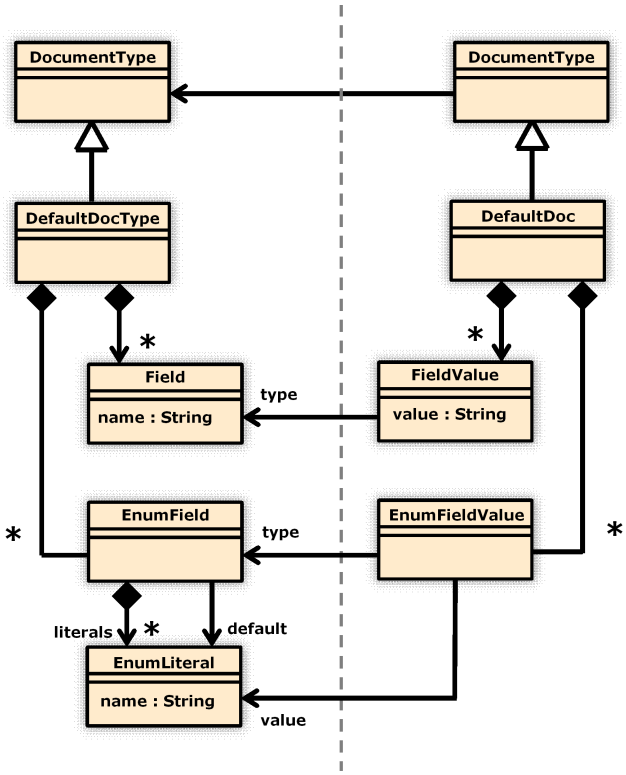

Figure 4.21: Information aspect: implementation of document types.

4.5.2.5 Conditions: Model

Refer to the model in Fig. 4.22. We integrate document conditions with two containment references from task: finish and start. Then we use ALL-coordinations to express that all start conditions have to participate in StartActivity and all finish conditions have to participate in FinishActivity.

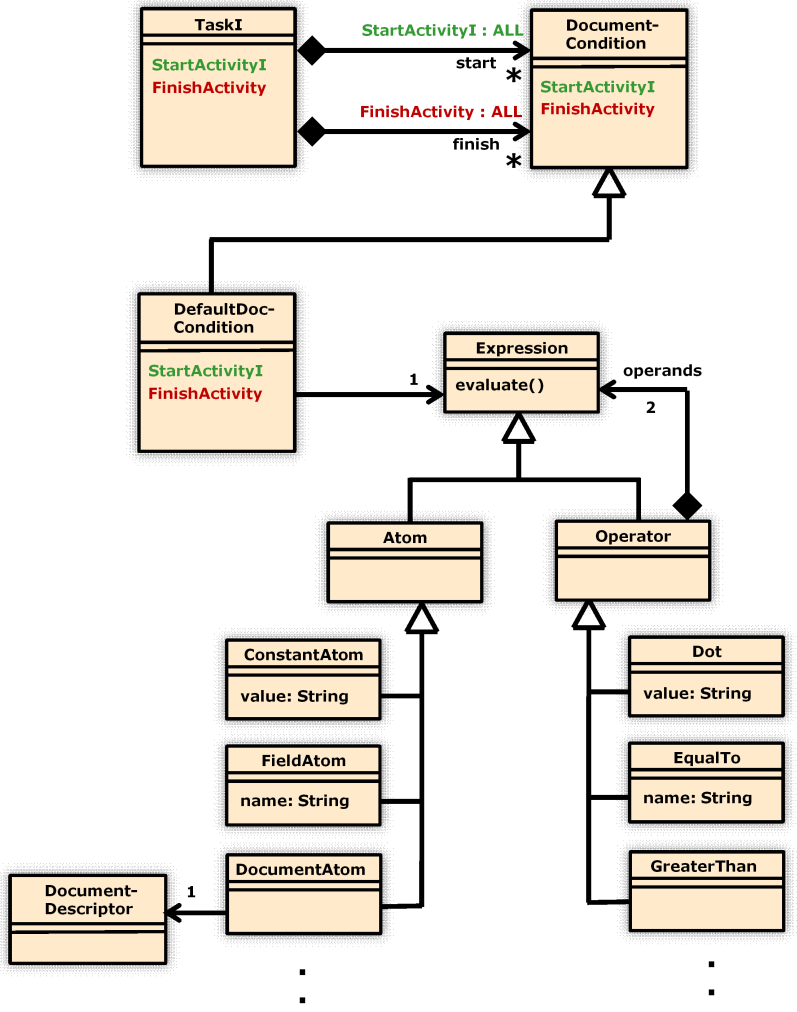

Figure 4.22: Information aspect: implementation of conditions.

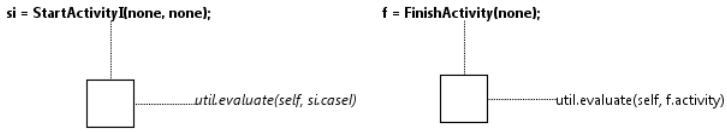

Figure 4.23: Local behaviour of default document condition.

The default model for document condition refers to an expression with a single operation *evaluate()*. We create a model for our expressions with an abstract syntax tree where we say that an expression can be an atom or an operator, and an operator has two expressions as operands. This allows for some degree of flexibility and nesting of the expression. In our implementation, atoms can be *constants*, *fields*, *documents* as included the figure, but also *enumeration fields* or *enumeration literals* which were omitted in the figure. Operations can be the the *dot* for referring to a field in a documents, but also *equal to*, *not equal to*, *greater than* and *less than*.

The evaluate method itself is coded manually without the use of ECNO. There would be no good reason to chose ECNO over Java code for this kind of operation. However, the evaluate method is invoked in the local behaviour of a default document condition element as shown in Fig. 4.23. The focus of this thesis is on how document conditions can be integrated with ECNO, we will not go into detail with the exact implementation of the evaluate operation but to say that we evaluate the abstract syntax tree recursively using an ad-hoc algorithm with fairly weak typing.

4.6 Realizing the Organisation Aspect

The model for the organisation aspect is simplified when compared to the previous aspect models. In addition we will present the implementing model directly - skipping the formalism independent model. We leave that analysis to future work.

4.6.1 Analysis

The AMFIBIA article does not present a model for the organisation aspect, however they state that the main concept is a *resource descriptor* which can

return assignments for a given task when it enables. We will here derive a model that is mainly based on Example 1, and a part of it can be viewed a simple implementation of a resource descriptor although that concept is not explicit in the models.

We want to be able to express in the organisation aspect that a given task requires a given role. For instance, in our example, Filter requires the role of Manager. This means our organisation aspect has to refer to roles and every task must have a reference to one or more roles which it can be assigned to.

Another concern we have to capture is that of task which follow up on other tasks, and therefore must be assigned to the same agent. For instance, Clarify follows up Submit in our example.

We will limit ourselves to modelling agents (no other resources), and we follow the convention of placing them in the dynamic model (refer back to Sect. 2.4.2.5). Finally, the model should be able to capture that agents at any given time are assigned to a number of activities, and that an agent at any given time have taken one or more roles.

Finally, we don't want roles and agents to be owned by the processes. Refer to the Workflow Reference Model (WfMC) where in Roles are part of an organisation model that is just referred to by the process definition. Likewise, agents are people, they only temporarily enact in a given process.

4.6.2 Model: Structure

In Fig. 4.24, we will start by expressing the the ownership of *roles* and *agents* in a global model, before we use these concepts in the process model. The *organisation* model is here very simple and just consists of roles, a real model would probably might have units and groups etc. We use a container called *people* for the agents.

In Fig. 4.25, please ignore the behavioural part at first. We model that the organisation aspect of processes are using certain roles and that individual tasks requires on of these roles. In order to support tasks the follow up on other tasks we use a simple reference *follows up* going from task and back to task. These concepts represents a simple resource descriptor implementation.

The dynamic model of the organisation aspect contains activity and case as we have seen before. Additionally it contains agents. Agents have a reference to the roles they have taken at a given point in time, and a reference to the activities

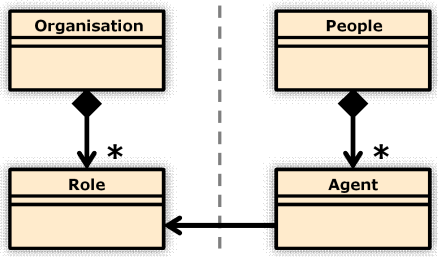

Figure 4.24: Organisation aspect: Global concepts.

they actively work one (have started). The reference is of the type opposite meaning that activities knows the agent they are assigned to. At last, cases have a reference to the agents who are involved in that case.

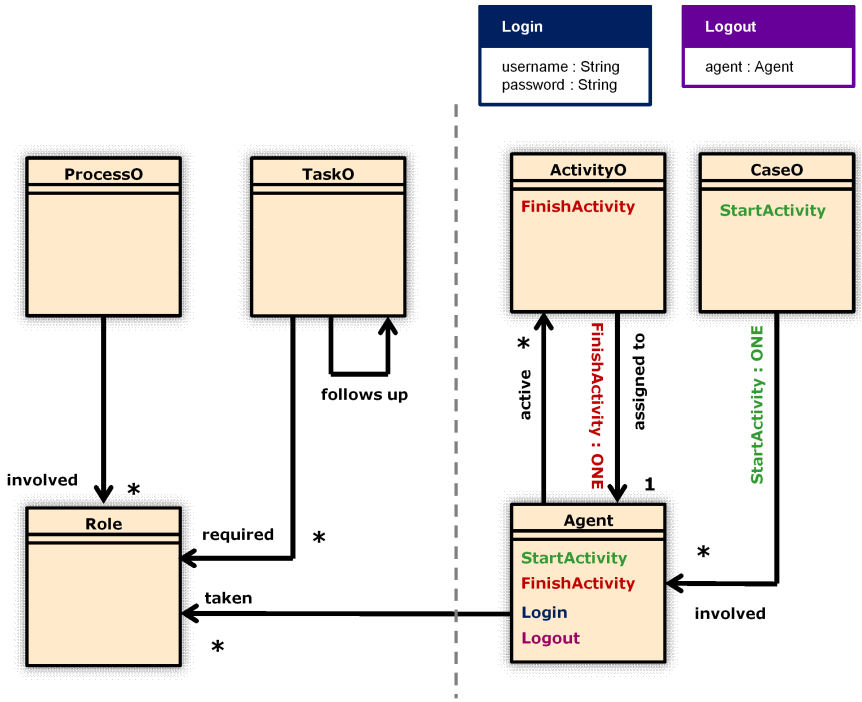

Figure 4.25: Organisation aspect: an implementation

4.6.3 Model: Global behaviour

Following the same pattern as in other aspects case StartActivity is coordinated though the case aspect, here it must find an agent that can participate, and we do that though the *involved* reference. FinishActivity is coordinated to agent through activity, to find one agent who can participate. There are two new events called Login and Logout, where Login takes user name and password as parameters and Logout takes agent as parameter. We will see in the local behaviour model of agent how this works out.

4.6.4 Model: Local behaviour

Please refer to the local behaviour of agent in Fig. 4.26. In the initial state an agent is *logged out*, from where he can participate only in a Login event. The parameters are expected to be contributed by the GUI controller. The event condition matches the supplied user name and password with the corresponding local attributes. After a successful login the state shifts to *logged in*. In this state an agent potentially participate in StartActivity, FinishActivity and Logout. FinishActivity does not have any further conditions. However, StartActivity has several conditions. First of all a match is performed against the GUI supplied agent parameter: the agent trying to start an activity must be the agent this element represents. Next the role of the agent is matched against the required role of the task parameter. Finally the follow up condition is evaluated in a utility method. The method checks within the case, if the task follows up other tasks, and if it does, it matches the agent that handled the other task against this agent.Logout is possible when the supplied agent parameter by the GUI matches a given agent element. A successful logout shifts the state back to logged out.

4.7 Discussion

Here we will explain the reasoning behind some of our design choices and discuss alternatives when appropriate.

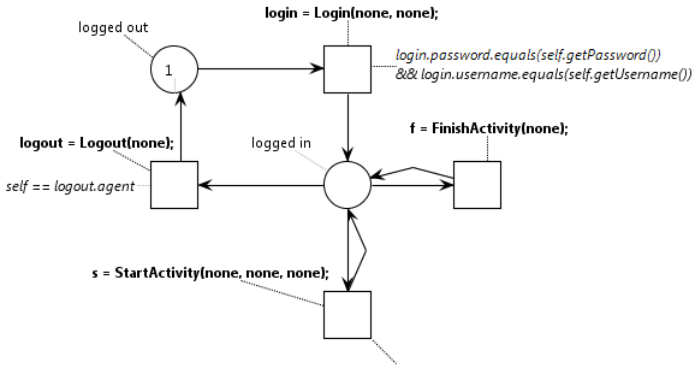

Figure 4.26: Local behaviour of agent.

4.7.1 Task identity

When looking at Fig. 4.7 a question may arise of why process and task participates in StartActivity, and secondly why we're not doing the same for FinishActivity. The answer is related to task identity. When doing StartActicity, task elements in the core will inject themselves as an event parameter, and task elements in the aspects (task aspects) will have a condition saying they must respectively be aspects of those core tasks. This prevents ECNO from computing interactions for StartActivity that involves more that a single task identity, which would obviously be wrong in this situation. Actually, the same should be guaranteed for FinishActivity (only one activity identity per interaction), but identity similarity is not at risk here because the coordination is attached to the reference from activity to its aspects. Still, there is an activity parameter, but it has a slightly different purpose, specifically it is a parameter that the GUI can input to "ask" if a given activity can finish (refer to Sect. 6 for detail).

4.7.2 Selection of trigger elements in the core

We trigger StartActivity in case and FinishActivity in activity. It has to due with the fact, that we also synchronise the aspect models with the core via case and activity. Some readers may wonder, if our implementation could have been more simple if we had synchronised aspects though tasks, since we could have avoided the task parameter mentioned above, in the discussion of "task identity". Perhaps it would have been simpler in some respects. But the choice we made

was mainly based on a navigation restriction. We can easily navigate from the dynamic model to the static model, but not so easily in the other direction. Therefore, it is easier to start "coordination trees" from the dynamic side of the model. Actually, we could navigate to case from task via not yet discussed top-level element containers in the implementation (*model registry* and *engine*). But, then we would have relied on elements that are not part our conceptual domain, which we did not like.

4.7.3 Building data structures in actions

When we create a new activity identity, we create four elements and link them. We could not find a way to distribute the "work" of creating a new activity identity on several participants of a single StartActivity interaction. The problem was that the newly created (core)-activity could not be shared with the aspects without injecting it into a parameter, which we preferred not to do - for efficiency reasons.

4.7.4 Duration of events

The main argument for splitting task execution into start and finish events is that events in ECNO, as in process algebra (CSP), must occur instantaneously - an assumption made at the theoretical level. In BPM, a task execution does have duration. Some work must be done and that takes a certain amount of time. Start and finish, however, can be treated as zero-duration events.

4.7.5 Agents as core concept

It is not ideal from a conceptual point of view that agents are known by the core, since agents belongs to the organisation aspect. This choice is a consequence of a need to input the identity of the agent who triggers StartActivity (used in organisation aspect) combined with no obvious alternatives to triggering StartActivity in the core. ECNO release 0.3.2 have the concept event extension which makes it possible for an event to inherit from another event while adding parameters. However, that concept does not appear to apply directly to this problem, because we still need input the agent parameter in case in the core. It is a potential topic for future work.

4.7.6 Instantiation of activities

In the analysis for the core model we introduces the instantiation concern. We selected to instantiate an activity only when is is started. We will explain the reason here.

The conditions for starting and finishing activities depends on complicated rules in each of the aspect models, and every time an event is executed the possibilities may change. Think of a process flow (control) where one of two execution paths can be taken. If one path is picked the other path is no longer enabled. Or in other words, certain activities which at one point in time could be started can no longer be started. It could also occur that a document needed for starting an activity is existing an one point in time, but is deleted before anyone actually started the activity. Generally speaking, we cannot be sure that resources needed to start activities will stay in the system after they have first arrived.

This leads to a problem that must be handled early on: When do we create new activity elements (objects)? If we create them when we discover an activity can be started, it could very well happen that we have to delete the object later because it has been made obsolete by other events in the system. Keeping track of which activities to keep and which activities to delete would be a non-trivial problem. This is why we selected to create activity elements when an activity is actually started. In other words, we define an event StartActivity which includes creating the activity element and includes starting it.

4.7.7 Instantiation of cases

Here we are turning the attention to cases, and more specifically to when we intend to instantiate them. When discussion the creation of activities we just assumed there already was a case to which new activities could be attached (refer to the structural model, activities are owned by cases). Actually there is, since we have selected to define a system invariant saying that all processes have exactly one counterpart case in *initiated* state. Cases can also be in *active* state. The transition from initiated to started shall then occur the first time an activity starts in them. At the same time, a new case shall be created to maintain the system invariant. This approach has several advantages: The user (agent) does not have worry about creating cases, it happens automatically. The rules for starting cases follows the rules for starting activities. More specifically, a case can only be started by starting the initial task of a process (in Example 1, this would be Submit). As a logical consequence, only the role who may start the initial activity (in Example 1, this would be Customer) may start a case.

We will now show how we achieve this in the models.

4.8 Summary

In summary, the behaviour modelling in this workflow engine aimed at capturing the events that takes place in a workflow engine, identifying the participating elements and the conditions for their participation. The life cycle of elements could be modelled locally using ECNO nets. In the actions associated to events we often make calls to external snippets of code, however, we used ECNO to integrate them at the right place and time, reducing the job of the coded methods to trivial factory operations or lookups in most cases. We used ALL-coordinations from core elements to aspect elements for the purpose of realizing aspect- (or subject-) synchronizations in relation to an event. This construct hides the technical details that we would have to worry about had we implemented aspect synchronization in the style of aspect oriented programming or in the style that ABFIBIA uses.

In the following chapter we will introduce the enactment GUI, and explain how it interfaces with the workflow engine realised by the models presented in this chapter.

Chapter 5

Enactment GUI

The models presented in the previous section realizes the workflow execution engine, and hence includes the design- and implementation concerns. However, the enactment GUI was not included in the models, therefore we will explain it here. In order to explain this, we need to talk about some ECNO concepts, which we only explained briefly in our introduction to ECNO. Specifically, the concepts which makes it possible to integrate a GUI.

5.1 ECNO's controller framework

ECNO makes it possible to define enable a GUI attribute in element types and event types in the coordination diagram. These are just configurations for ECNOs build-in GUI which can be used to test the generated software. In our case, the build-in GUI is not sufficient for making a convincing workflow enactment GUI for our workflow engine. Therefore, we had to use ECNO's controller API. We will now explain main features of this API.

5.1.1 Element Event Controllers

The purpose of ECNO's *element event controllers* is to allow applications (could be in the GUI part) to get information about the possible interactions in elements, and to execute them. This can be used for updating the GUI accordingly.

An element event controller knows about an element and an event. For example, an activity element and a FinishActivity event. When initialized, and when *updated*, the element event controller automatically calculates the possible *interactions* for the event, in that element, and keeps the interactions in a local *interaction iterator*. If there is at least one interaction in the iterator, the element event controller is enabled. An enabled element event controller can execute the next interaction in the iterator, when it is requested to. At last, an element event controller registers an *invalidation listener* on all the interactions it currently holds. It will receive a call-back if any of these are invalidated, and perform an update. An interaction is invalidated if one or more of its participating elements are changed.

In the following section, we will explain how we can use an *engine controller* to create the element event controllers dynamically.

5.1.2 Engine Controllers

The purposes of *engine controllers* is to allow applications to receive information about, when elements are added or removed in the runtime information, and to allow applications to create element event controllers for the added elements.

The ECNO engine sends an *add element* notification to all registered engine controllers when it becomes aware of a new element. This already implies, that an element, which has been added to the runtime information is not automatically added to the engine. Adding an element to the engine shall be done manually by the applications by calling a method on the engine.

When engine controllers get notified of a new element (for instance, an activity) they can decide how to react to that. A GUI would normally check the type of that element and then display some graphical representation, or it could just ignore it. If the added element is relevant, an engine controller would typically create an element event controllers for it.

Engine controller are registered with the ECNO engine. It can happen when the application launches or at a later time.

5.2 Maintaining GUI lists of interactions

The main features of the ECNO controller API have now been explained. However, we have an additional problem, which cannot be solved with element event controllers. Recall from Sect. 4.3.3, that activities are created and started by the same interaction. for example, if we want to present a list to the end user showing the activities he can start at a given time, we have to list all the possible interactions of the type StartActivity. The reason, that we cannot use element event controllers for this, is that it can only be used to execute the next interaction in the internal iterator it keeps, not to maintain a list of interactions in the GUI. This means, we have to manage the interactions manually, for this purpose, instead of letting the element event controller manage them for us.

In order to accomplish this goal, we can call a dedicated method *getInteractions(..)* on the engine, passing an element and an event as parameters. The method will return an interaction iterator containing all possible interactions. In the example, we have to pass the element case, and the event StartActivity. We have to do this for all the cases we are interested in. Actually, the main difference from using element event controllers is that we now have access to the interaction iterator, which is private in element event controllers.

With this technique, we can list all the possible interactions in the GUI. We can also execute any of them when the user wants to. This comes at a price of added complexity, since we have to handle the invalidations manually too - e.g. update the GUI when interaction are invalidated. We will elaborate on this aspect in Sect.5.6.2.

5.3 ECNO Connectors

Here we will introduce a concept that we added in order to make GUI integration a bit more convenient. ECNO connectors are extensions of element event controllers, which means they inherit all the features of element event controllers. In addition, they have methods to add and get event parameters. ECNO connectors know the GUI that created them, and after an update, they will make a call-back to that GUI, to inform if they are enabled or not. Later in this chapter, Fig. 5.4 will indicate how ECNO connectors are used.

5.4 The Enactment GUI

We will now continue by analysing what is required of the enactment GUI. The main purpose of our GUI is to demonstrate that the workflow engine is working as it should, but it should also give us the feel of a workflow engine when using it. We aim to create a GUI that follows the patterns defined by WfMC (refer to Fig. 2.1). This implies agents shall be able to log in and see a personalized view of tasks (the worklist). We will combine login and worklist viwing in a single GUI component, that we call the Worklist Viewer. When an agent wants to perform the work of a task, he shall be able to open it, and view input documents, and view, create or edit the output documents. We will call the view of an open task the Task Viewer.

In this chapter, the focus will be on the Worklist Viewer, because this is the interesting part from an ECNO point of view. The Task Viewer is really nothing but a viewer and an editor for documents.

5.5 Worklist Viewer

In the Worklist Viewer we mainly want to maintain two lists. One list showing the possible tasks that an agent can start (Inbox), and another showing the started tasks (Work In Progress). We note, that the tasks an agent can start are actually the same as the combination of all StartActivity interactions in case when the specific agent is injected as a parameter. Likewise, we note that the work in progress is the same as the activities that are linked to the agent in the runtime information (refer to the runtime information model of the organisation aspect in Fig. 4.25).

The ideas for the Worklist Viewer mentioned above can be seen in the form of a principle drawing in Fig. 5.1. In the figure, we can see the Inbox and the Work In Progress list. The content (list items) is compatible with Example 1, meaning that Jack, the Customer, is logged in, and we see his tasks.

5.5.1 The Inbox

We first explain how the list is populated, then we explain how it is used to start an activity. In order to populate the Inbox, the GUI iterates over all cases that Jack is involved in, and for each case it gets the StartActivity interactions

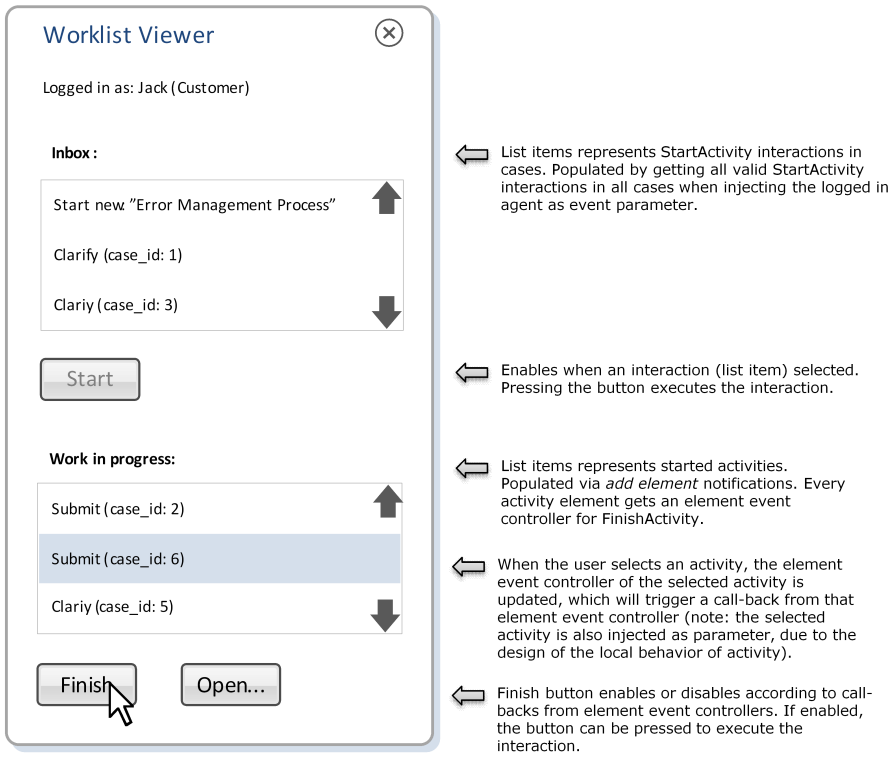

Figure 5.1: Principle drawing of Worklist Viewer (login features omitted).

when the agent is injected as parameter, and adds them to the list. This is the technique of maintaining interaction list in the GUI, that was explained above. We can enable the Start button when any of the interactions in Inbox is selected. The selected interaction can always be executed, or it would not be an interaction, however we have to respond to invalidation notifications and re-calculate the interaction to ensure this list is always valid. Finally when the Start button is pressed, we can call *execute()* on the selected interaction. We have included a diagram showing what happens dynamically when the user selects an interaction and starts an activity, please refer to Fig. 5.2. As can be seen, the steps are very simple. With the Inbox, the complexity lies in keeping the list updated, and with a minimal performance loss. We discuss this issue later in the chapter.

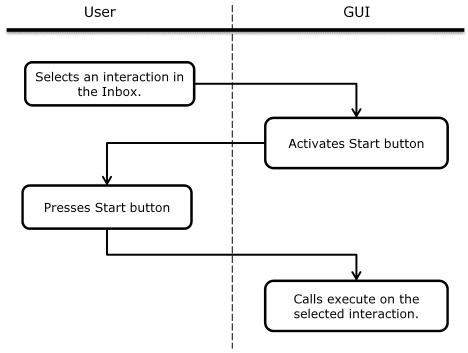

Figure 5.2: Starting an activity from the GUI.

5.5.2 The Work In Progress

We first explain how the list is populated, then we explain how it use used to finish an activity. The Work In Progress list is populated, at login, by finding all the activities in the runtime information which is assigned to Jack. Activities which are created later, can be caught with the engine controller notification system (*add element*). For every activity, we register an element event controller (ECNO connector) allowing us to hook into the status of its FinishActivity event. When the user clicks on an activity, we update the corresponding ECNO connector [1], which then makes a call-back to the GUI to inform about its enabledness status. The GUI will update the Finish button accordingly. When the user clicks on an enabled Finish button, we call *execute()* on the ECNO

[1] The activity is injected as parameter first, because the local behaviour of activity requires it - this is redundant now, and could be designed out.

connector (hence, element event controller) of the selected activity. We have included a diagram showing what happens dynamically when the user selects and finishes an activity, please refer to Fig. 5.3.

The button Open enables whenever an activity is selected, and when pressed, the Task Viewer is launched. We have implemented a Task Viewer GUI but we omit a discussion in this report, because it is a secondary topic in relation to ECNO.

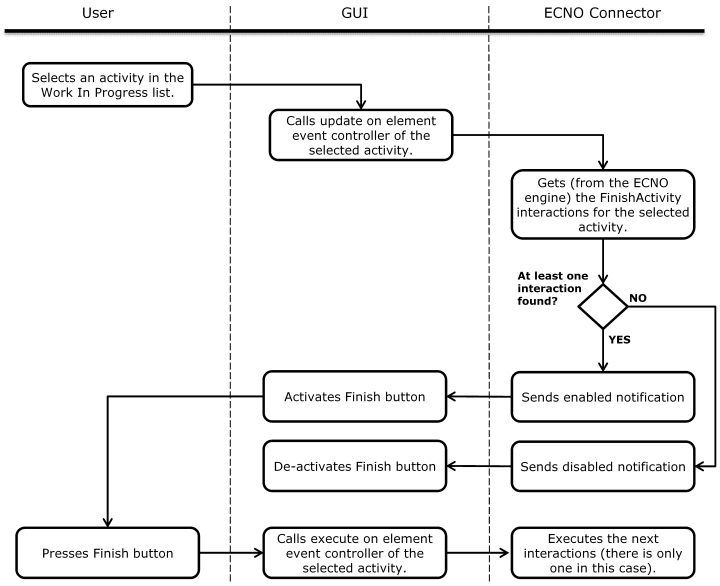

Figure 5.3: Finishing an activity from the GUI.

5.6 Design and Implementation

We have explained the ECNO concepts for integrating a customized GUI, and shown the principle of how we use them to integrate the Worklist Viewer GUI, of which we have also seen the principle layout. The purpose of this section is be more specific on how these different constructs were realized, and we will also discuss some performance related topics. This section combines design and implementation concerns.

5.6.1 Design of Worklist Viewer

The remaining problem is, that the individual GUI components, such as lists, list models, and buttons, need to be integrated with the ECNO components, such as element event controllers and interactions, and with particular elements of the runtime information displayed by the GUI. Here, we will present the structure we use in the Worklist Viewer to solve this problem.

Please refer to the class diagram in Fig. 5.4. First notice colour usage to categorize the classes and interfaces. The blue colour indicates ECNO classes and interfaces. Green colour is used for Java Swing classes. Java Swing is the name of the framework that we have used for creating the enactment GUI. Other classes are the default yellow. We will now comment in the structure in more detail.

Recall that the ECNO framework for controllers includes engine controllers and element event controllers. In the figure, these concepts are represented by the interface IController and the class ElementEventController respectively. The IController interface is implemented by our WorklistViewer class, which represents the Worklist Viewer introduced earlier. The WorklistViewer class extends JFrame of the Java Swing framework which makes it a GUI component. WorklistViewer owns the buttons start and finish and two Jlist objects representing the Inbox and the Work In Progress. The models of the JList objects contains objects of the type ElementItem in the the case of the Work In Progress list, and objects of the type InteractionItem in the case of the Inbox.

An ElementItem keeps a reference to the element (for instance, an activity) and to the ElementEventController object which was created for that element. Notice we use the extension of ElementEventController called ECNOConnector, where we added some convenience methods for adding and getting parameters. ECNOConnector is also able to make a call-back the the GIU that created it when it enables or disables (being an ElementEventController, it can enable and disable).

An InteractionItem just keeps a reference to an interaction. ElementItem and InteractionItem overrides *toString()*, which returns the text we want in the list. In the case of an InteractionItem, this text will be different if the activity, which can be started is the initial in a case. This will let the agent know, the action will also start a new case (Fig. 5.1 shows an example of this).

The Start button is enabled whenever an InteractionItem is selected in the list, and when the Start button is pressed the select interaction is executed. The Finish button is enabled or disabled by the WorklistViewer according to call-

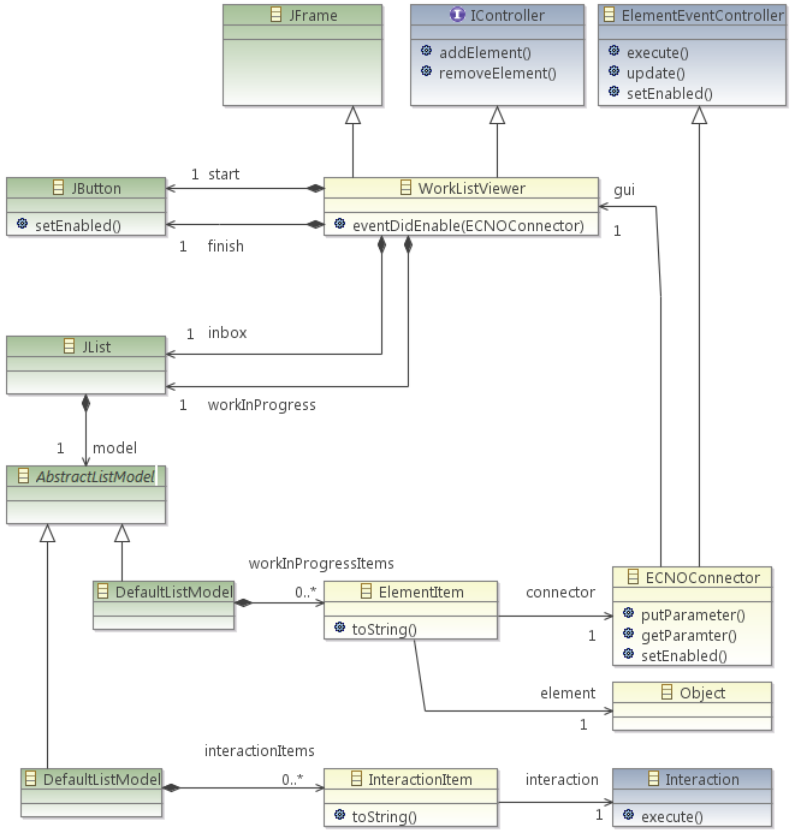

Figure 5.4: Class diagram: Worklist Viewer

backs (to *eventDidEnable()*) from the ECNOConnector objects. Recall that ElementEventControllers receives notifications (to *setEnabled()*) when their event enables or disables.

5.6.2 Performance optimization

Here we discuss a performance optimization in the enactment GUI, focussing on invalidation notifications for interactions, which were found to have a significant impact on the systems responsiveness, and was found to dominate other possible effects for the moment.

5.6.2.1 The problem

Recall, from the discussion of working with interactions directly, that we mentioned a topic of interaction invalidation. It is that mechanism, we wish to optimize for our application. When populating the Inbox with StartActivity interactions, we request ECNO to calculate interactions in each case individually with *getInteractions(..)*. Interactions are returned by the ECNO engine in a collection called an *interaction iterator*, and then the GUI registers on this iterator as invalidation listener. An interaction iterator sends an invalidation notification when any of the elements used in the calculation are changed. Whenever this happens we need to update the Inbox. However, we cannot update the entire Inbox, that would be too inefficient: if we updated the entire list at every notification, the running time would be proportional to the number of cases, a solution which would definitely not scale. Therefore, we only wish to update the part of the list corresponding the the particular iterator (e.g. in one case) that was invalidated. In the following we will explain how this was implemented.

5.6.2.2 The solution

When an iterator invalidates, the notifier (the iterator) passes the iterator (itself) that was invalidated to the listener (our GUI). However, in the current ECNO release, the listener cannot get the case element the iterator was based on from the iterator. Therefore, we maintain a map in the GUI, which takes an iterator as key and returns *iterator information* as value. Iterator information contains a reference to the case we calculated the iterator in, as well as references to all the associated list items we placed in the Inbox model for that iterator. When an invalidation notification is received, we take the iterator, and use the map to

get the list items, which we then remove from the list. Then we use the map again to get the case. Then we request the ECNO engine to re-calculate the interactions, only in this case, and we add the result to the Inbox model.

5.6.2.3 Evaluation

This optimization had a huge impact on the performance. FinishActivity interactions improved from linear to constant time complexity, in terms of number of cases. StartActivity interactions improved from quadratic (assumed) to linear time complexity. In both cases one factor, the number of cases, was taken out. As we will show in the Evaluation chapter, a performance issue still remains for StartActivity. Just briefly, the GUI is getting excess invalidation notification from interaction iterators. It could not be solved with this version of ECNO (v0.3.1), so new concepts are needed in ECNO for this purpose.

5.6.3 Summary

In this section we have explained how the Worklist Viewer was implemented and discussed an important performance issue. Our enactment GUI includes two other components, the Task Viewer and the Database Browser dialogue, but they do not interface directly with ECNO. They are implemented with standard GUI programming techniques and therefore we do not discuss their realization in detail in this report, instead we refer to the provided source code.

We further refer the reader to the Chapter 7, which includes screen shots of all three GUI components in action (Fig. 7.5, 7.6, 7.7 and 7.8).

Chapter 6

Implementation

This chapter is about the the more technical aspects of our project. What is left, is to show the actual models we used to implement the workflow engine, to explain how we generated a simple tool for business process modelling (the process definition tool), and to show how we integrated the runtime information. When referring to business process models and runtime information, are still using the terminology developed in Fig. 4.1.

6.1 Workflow Engine - actual models

The purpose of this section is to show how we implemented the workflow engine using EMF Ecore models and ECNO models. We only show an example, which demonstrates that the ad-hoc notation we used in Chapter 4, maps directly to the actual models.

Please refer to Fig. 6.1 showing how the ad-hoc notation we use in Chapter 4 is in fact the combination of an Ecore diagram and an ECNO coordination diagram, except we do not print the class attributes in the ad-hoc notation. Note that, the ECNO nets we used in Chapter 4, were the actual diagrams.

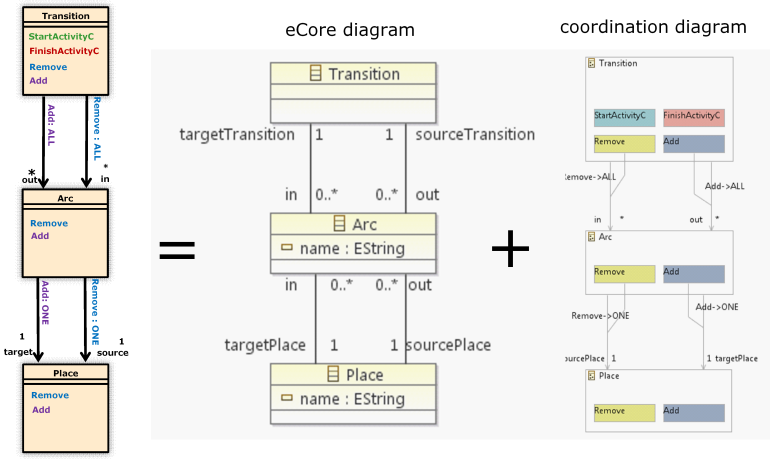

Figure 6.1: Ad-hoc notation vs. actual models.

6.2 Process Definition Tool and Runtime Information

The process definition tool can be used to create and edit the business process models. This section explains how it was realized as an EMF editor, based on our meta-model for business process models (the one presented in Chapter 4). First, we will explain what an EMF editor is, since some readers might not be aware of that, and then we will explain how we used the technology to create the process definition tool, after some initial concerns were dealt with.

6.2.1 EMF editors

The Eclipse Modelling Framework (EMF) comes with a code generator, which can generate a simple tree editor based on an Ecore meta-model. An Ecore model can be described as EMF's version of a UML class diagram. By default, the an EMF runs in the Eclipse runtime-workspace as a plug-in project. Using this editor, the user can create models that conform to the meta-model, and exists as *resources* in the Eclipse runtime-workspace.

6.2.2 The goal

We will now present, what we wanted to achieve (the result), and in the following section we will explain how we achieved it. In Fig. 6.2, an example of the resulting project folder in the runtime-workspace is shown, with example of a resource opened with the EMF editor (the process definition tool).

Notice there are many separate resources in the leftmost part. All business processes are placed in their own resource. We see a *model registry*, where we can register (refer to) the processes. There is a resources named *engine*, which contains all the runtime information. Runtime information can can refer to the process models. The engine and the model registry both refer to each other. We have resources containing the concepts that are not owned by processes, namely *global resources* containing roles, agents, document types and documents. The business process models, can refer to roles and document types. The runtime information can refer to all global resources.

The rightmost part is showing an open resource, actually what you see is a *resource set* containing the resource that was opened, but also the resources that are *imported* (referred to) in that resource. In the example, we can see that when opening the resource containing the Error Management Process (Example 1), the displayed resource set includes the resources the opened process model refers to. In particular the referred resources contains the externals organisation model, the book store organisation model, a document type model and the model registry. Within the process model itself, the core model and the individual aspect models conforms to the meta-models presented in Chapter 4. Note that these meta-models were presented with our ad-hoc notation, which included ECNO behaviour, but technically, they all have Ecore diagrams underneath.

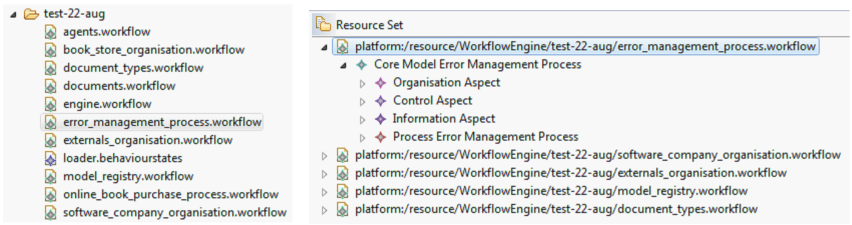

Figure 6.2: Process Definition Tool (EMF editor).

6.2.3 The problems

The above approach sounds quite easy, and seemingly we could just generate an editor based on our meta-model for business processes using the EMF generator, but we had have a few problems for our process definition tool which we needed to take special care of.

Problem 1) Every part of our business process meta-model (and runtime information model) that we want to create (technical) instances of in a separate resource in the runtime-workspace must have a suitable container for that part of the model. Hence, we need a container for business process models, we need a container for runtime information and we need a container for each individual process. In general, a suitable container class may or may not be in the meta-model. In our case, we found that we had to add container classes just for this purpose to get the result we wanted. We will show which ones in the following section.

Problem 2) Another concern, which follows from this, comes from the fact that EMF does not handle cross resources containments very well (for technical reasons). We should therefore make sure, that in our meta-models, we avoid containment of classes where instances shall be placed in different resources. In the following section we show the solution we have made.

6.2.4 The solution

Here, we will present an Ecore mata-model adapted to above mentioned problems. From this model, EMF can generate a process definition tool (an editor) that operates on business process models in separate resources. However, the same Ecore meta-model can still be used underneath our ECNO models, when generating the workflow engine (the enactment environment).

As can be seen in Fig. 6.3, we added *engine* as container for the runtime information. *Core model* is a container for a single business process model (with aspects), while *model registry* is a container for core models. At last, we added the interfaces *global aspect* and *global runtime aspect*. These interfaces can be implemented by aspect concepts that are not part of processes. In particular they are implemented (in other Ecore diagrams) by containers for documents, document types, agents and roles.

Notice that certain places where containments would have been expected we are using bidirectional references, to avoid the problem of cross resource contain-

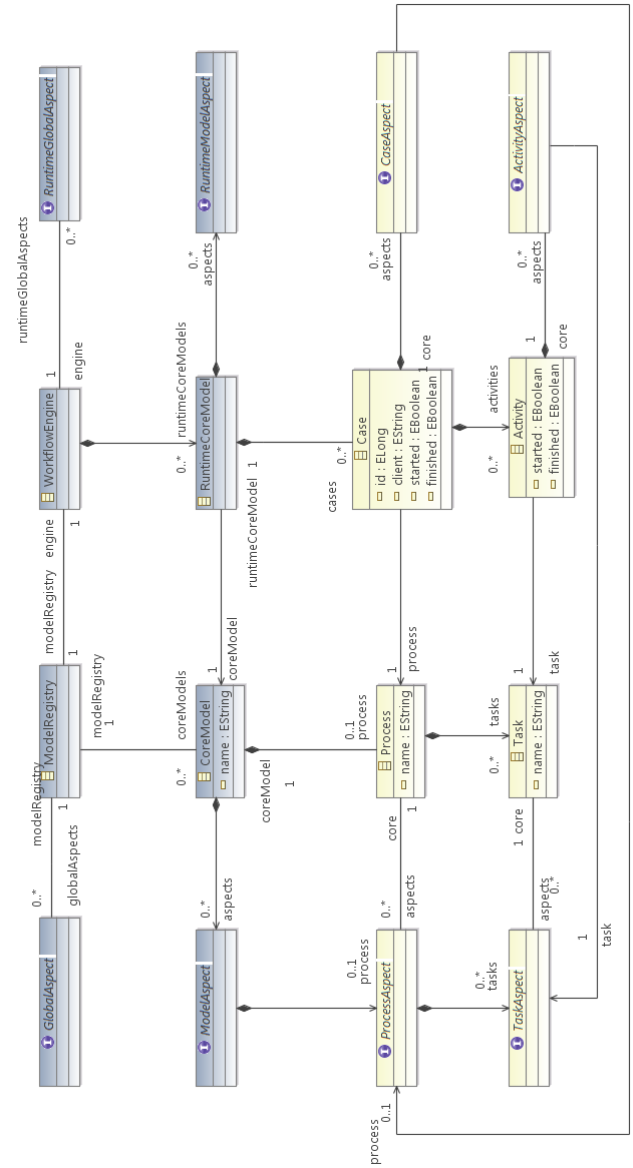

Figure 6.3: Technical model layers added on top of conceptual model. This figure has the lower layer in common with Fig. 4.5 and Fig. 4.7.

ments. The classes that are not contained anywhere (engine, model registry, core model, global aspect, runtime global aspect) will be owned (contained) by resources in the runtime-workspace when the user creates the models.

With the bidirectional reference between the model registry and the engine, it would seem we have introduced an exception to the rule that we only refer from runtime information to process models. However, conceptually the engine is not a part of the runtime information, it just contains it. The reference is needed for navigation when the enactment environment creates the runtime information based on the models. We will elaborate on how this works in the following section.

6.3 The behaviour-state resource

We still have a couple of concerns that have not been covered yet. In particular, how we launch the enactment environment and how we persist our runtime information. Additionally, the local behaviour models were modelled with ECNO nets, which in general defines different states for an element. These states are not a part of the runtime information as we have defined it, so we need to explain where the runtime states are kept. The build-in ECNO features, that we will explain in this section, are addressing the above mentioned problems.

Notice in Fig. 6.2, that the runtime-workspace contains a resource of a different type that the rest - e.g *loader.behaviourstates*. A behavior-state resources contains concepts defined in ECNO, and the file is maintained by the ECNO engine, except we have to set up the initial version at first. The resource imports the runtime information resource (and hence the resources the runtime information imported), and contains references to all elements the engine is aware of at a given time (recall, element are added to the engine by calling *add element* on it). For each added element, the state of the local behaviour is kept.

ECNO plugs in an action in the Eclipse IDE named "Launch ECNO Engine". This action can be started by the user from behaviour-state resources. When the action is executed, the ECNO engine launches, and then it will run a method in an application class, referred to in the behaviour-state resource. In this application class we launch our GUI instances (Worklist Viewers) and register them with the ECNO engine as engine controllers. The ECNO engine will then send an *add element* notification to the engine controllers. This realizes loading of elements and their states at start-up.

The last topic is that of persisting the runtime information (and the states).

Actually this is easy because everything is contained in Eclipse resources, which are supporting serialization. We let another ECNO tool (*ECNO: Engine Registry*), which is also plugged into the Eclipse IDE by ENCO, handle this part. The tool is realized as an Eclipse View and contains a list of started ECNO engines and a button which looks like a floppy disk. When this button is pressed, the tool saves the content of the behaviour-state resource and the contents of the referred resources.

6.4 Development workspace

In this section we will show how our project was structured in the Eclipse development-workspace, and we will also show how the actual global behaviour models looks like since we have used an ad-hoc notation earlier in this report.

6.4.1 Project Structure

Please refer to Fig. 6.4 showing our development-workspace for the workflow engine Eclipse project. As can be seen, the models folder includes a number of *Ecore* diagrams, of which we have seen *WorkflowInfrastructure.Ecorediag* in Fig. 6.3. The other three are the models of the individual BPM aspects we support. The diagram in *Workflow.ecno_diagram* contains all global behaviour - ECNO release 0.3.1 did not allow us to split that diagram into aspects. The file *Workflow.pnml* contains the local behaviour models. In the folder *scr* we have the generated classes (emitted by EMF and ECNO generators). In the folder *coded* we have the classes that are coded manually. In particular the GUI and the code snippets. The seperate packages in the lower part of the figure contains the generated EMF editor code.

We have aimed to avoid modifying generated code and the was actually achieved. Regarding ECNO generated code, we never had a situation when we needed to modify the code. With the EMF generated code it was different. In the EMF code, we had to omit constraints on which child elements can be created in the static instance models. As a consequence, the editor allows creating child elements under the control aspect model that belong to the organisation aspect model. Since we're not building a system for real users, but rather a test platform, we did not implement the child element constraints, but did leave an example in the project tree showing how to do it.

Above mentioned compromises have so far proven to be a good choice. We have

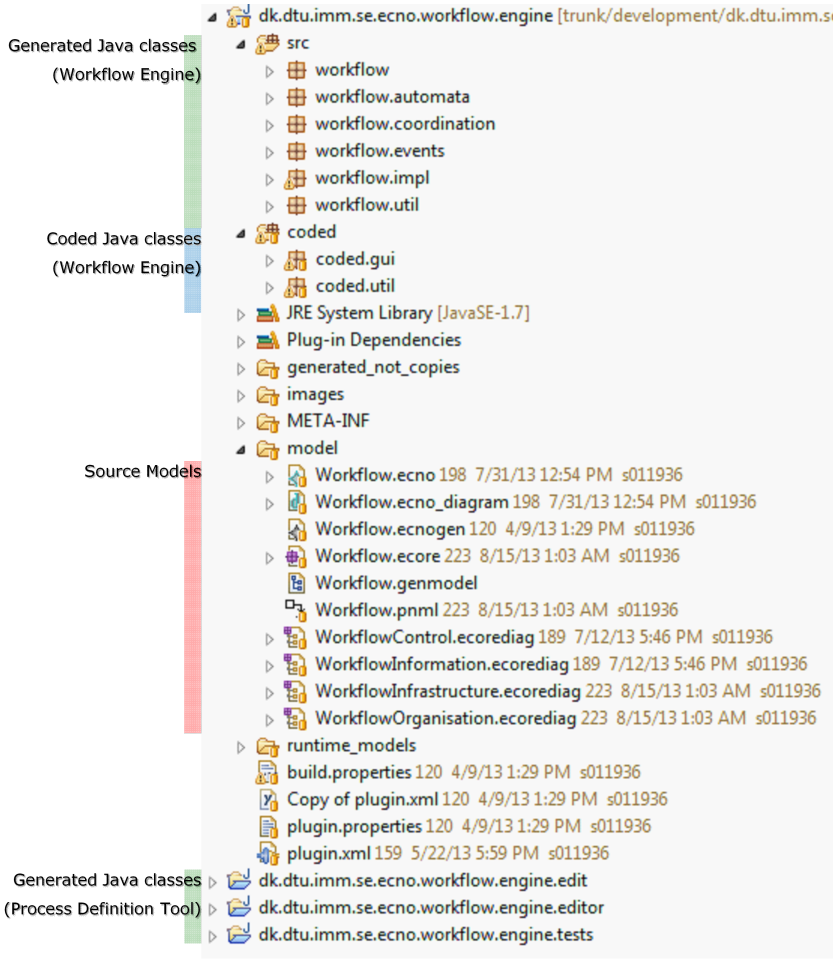

Figure 6.4: Development workbench.

made several improvements to models (even at a very late stage in the project) by deleting all generated code, updating the models and re-generating the code. These improvements would not have been feasible (time-wise) to implement if we had customized the generated code. While maintainability is an important quality attribute of a workflow engine we would stress that avoiding to customize in generated code is a very good principle.

6.5 Summary

In this chapter, we have discussed the technical aspect of our project. In the following chapter, we will carry out acceptance testing to prove that the system is working according to expectations.

Chapter 7

Acceptance Testing

In this chapter, we have the intent show that the workflow engine implementation works as expected. Since this is a scientific project we have taken a light weight approach to defining the requirements and we will limit the testing to acceptance testing. Acceptance testing means that we test from the user perspective, we don't test all individual features in details. The use cases for the acceptance test cases have been derived from the chapter Project Scoping.

We have also carried out initial performance testing but we present these tests in the the chapter Evaluation, because we do not test against pre-defined performance criteria (that was not the intention with this thesis) and because we want to present the results and the evaluation in one place.

The process definition tool is a simple tree editor generated with EMF, so we will not test it independently. It is tested indirectly by acceptance testing based on Example 2.

Our test strategy is that we will create a new example (Example 2) to use in the acceptance testing. With this approach we avoid the risk of undetected errors as a result of using the same example in the testing as was in development.

7.1 Example 2: An Online Book Purchase Process

Here we will explain the example process of a book purchase scenario, but in a little less detail than the explanation given for the previous example process. At this point the reader will probably be familiar with the notation in Fig.7.1 and Fig. 7.2 with which we define the process, if not refer back to the walk through of Example 1 in Sect. 2.3, where also the concept of input and output documents is explained in detail. Recall that, the dotted lines are just an ad-hoc notation to indicate that a task has a start and/or a finish condition (see also how this was realized in Sect. 4.5.2.4) referring to fields in a particular document. If the arrow is going from a document to a task, the document is an input document to that task. If the line is dotted, there is one or more start condition(s), where the conditions are written next to the line. If the arrow is going from the task to a document, the document is an output document. If the line is dotted, there are one or more finish condition(s).

Just briefly, the process defines that a Customer can place a book order, which a Librarian will then take. The Librarian will then check if the credit card information given in the *payment* is valid and if the selected *book* is on stock. The Librarian will create a *status* document, which contains the information about whether a book was on stock or not in stock, and if the credit card was accepted or rejected. If the payment is rejected the order will be rejected by the Accountant. If the book is not in stock the order will go into Back Order where a Buyer will get the book from the supplier. When the book is on stock the Shipping Agent will ship the book to the Customer and produce a *receipt* with the shipping information which the Customer can view. At any time after the Librarian has processed the order, the Customer can Track the status of the order.

A few details have been omitted in the Fig. 7.2 for the sake of presentation. For example, Track and Check Receipt have follow-up relations to Place New Order - e.g. only the same Customer can take the respective tasks.

The control process (Fig.7.1) of Example 2 has a feature that Example 1 did not have. The process branches out after the Take Order task. Our Petri net notation expresses this easily, and since the controls aspect of our workflow engine is implemented using Petri net semantics it should be a problem to execute it.

When looking only at the control process, the task Back Order might look a bit dangerous because the process could get stuck here if the task is started every

time it can be started. To see how we resolve this in the information model refer to Fig. 7.2: The Back Order task can only start if the book was not in stock, as expressed with the condition. And the Back Order task can only finish when the book is back in stock, and then it cannot start again, the process will continue with the task Ship.

Apparantly a case will never end, becuase the token in the input place of Track will always be placed back when Track is finished. To overcome this, we are per definition always finishing a case when a token reaches the place named finish.

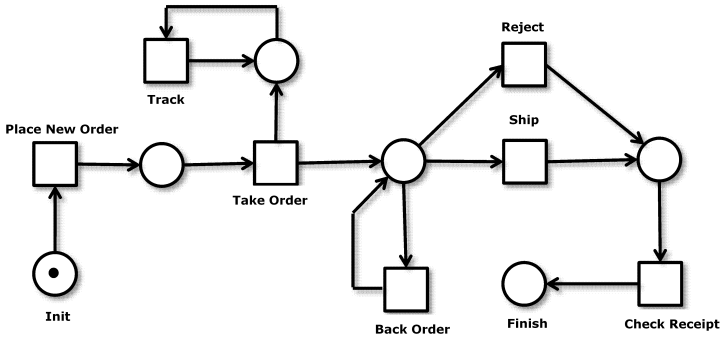

Figure 7.1: Example 2: Control aspect of an online book purchase process.

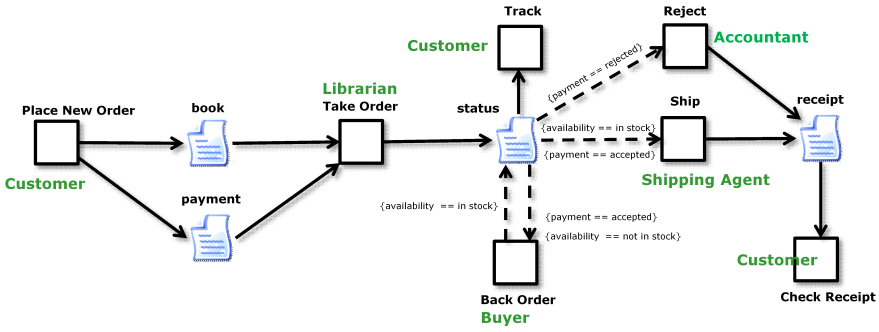

Figure 7.2: Example 2: Information (and Organisation) aspect of an online book purchase process.

7.2 Building the model

The models that we have just seen were created by the process definition tool. While the tool is just a generated EMF tree editor, which conforms to the structural meta-model of our workflow engine, there is no need to show the whole tree in this report. Figure 7.3 should be enough to give the reader the basic idea of how this looks. The full example model is included with this thesis hand-in.

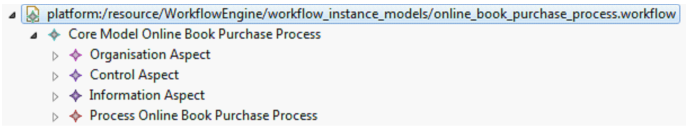

Figure 7.3: Example top level view in process definition tool.

The user interface is not that user friendly. It does not matter at all for the purpose of this project, because it has no importance in relation to the concepts that we try to demonstrate, but it would be a nice continuation to create graphical editors with model validation support. We leave that topic to future work.

7.3 Scenario testing

We will here explain the execution scenarios which have been tested. The text includes references to screen shots taken during the actual test. First we will present the agents that have been created for the purpose. The agents are listed in Fig. 7.4 with their role. Each role is defined in an organisation model, so we list also the organisation the role is defined in. Notice that Customer is defined in the Externals organisation. Example 2 as well as Example 1 involves this role. As a consequence, Jack, who is a Customer can engage in both processes.

7.3.1 Scenario 1 (case 1): Purchase which goes through

Jack logs in and starts a new purchase process. He opens Place New Order and selects a book from a list of pre-existing book documents (Fig. 7.5) and enters his credit card information (with New Document). Jack finishes the task. Ellen, the Librarian, is already logged in and receives the task Take Order in

Agent	Role /Organisation	Username	Password
Jack	Customer / Externals	brian	pw
Ellen	Librarian / Book Store	ellen	pw
Tom	Buyer / Book Store	tom	pw
Max	Shipping Agent / Book Store	max	pw
Anton	Accountant / Book Store	anton	pw

Figure 7.4: Agents.

her inbox. She starts it, and opens the task, where she creates a new status document. She sets the field Availability to the value In Stock, and she sets the field Payment to the value Accepted (Fig. 7.6). Jack receives the task Track in his inbox. The Shipping agent, Max, who is already logged in receives the task Ship which he starts and opens (Fig. 7.7). He creates a new receipt document and finishes the task. Jack receives a Check Receipt task. When Jack finishes Check Receipt the case ends.

7.3.2 Scenario 2 (case 2): Book is unavailable

Starts just like scenario 1, but Ellen sets the field Availability to the value Not in Stock (and accepts the credit card again) before finishing the task. Bob who is already logged in receives the task Back Order, he opens it, and cannot finish the task until he sets the field Availability to the value In Stock (Fig. 7.8). Now, Max can ship the order just like in scenario one.

7.3.3 Scenario 3 (case 3): Credit card is rejected

Starts just like scenario 1, but Ellen sets the field Payment to the value Rejected. Now, only Anton received the task Reject where he can create a receipt saying that the order was cancelled. The scenario ends like scenario 1.

7.3.4 Conclusion

All the above scenarios have passed according to the behaviour expected from the process definitions. Note that even though this was not a specific scenario, the workflow engine has also been proven to work with with several processes (of different kinds) at once. Notice also from the screen shots that Jack is the only "agent" who (since he is a Customer) is involved in cases of the type "Error Management Process" which was the same as Example one. We have as well tested with other agents in Customer roles, and checked that they cannot follow up on tasks handler by other Customers.

Note: The following pages will show the screen shoot we referred to, and after these figures we discuss the GUI test cases.

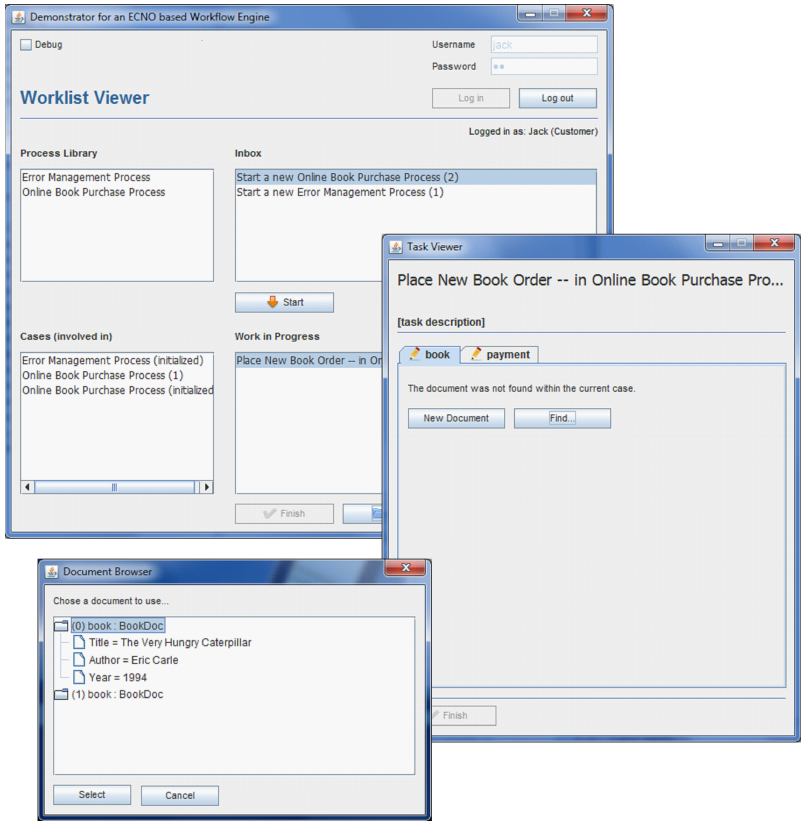

Figure 7.5: Scenario 1: Jack (Customer) selects a book to order.

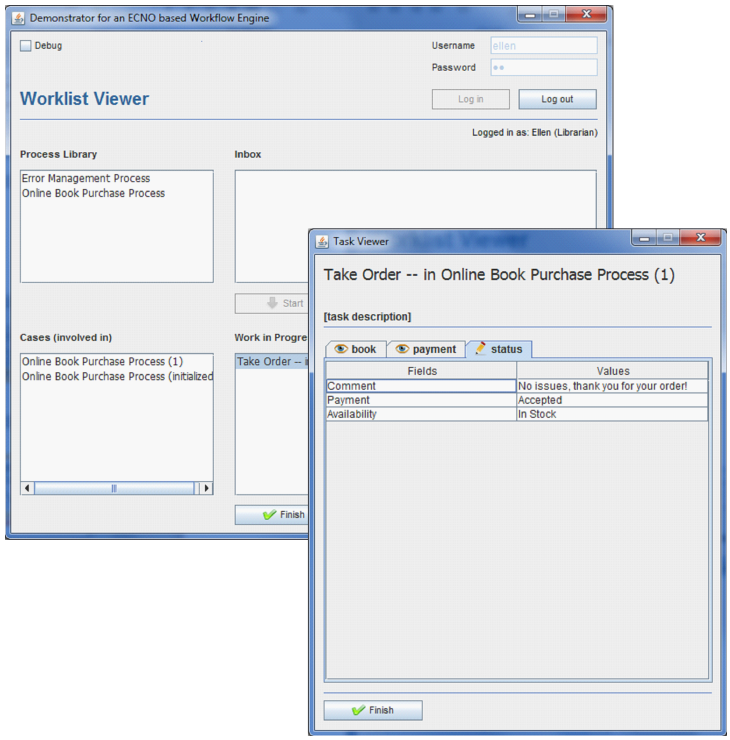

Figure 7.6: Scenario 1: Ellen (Librarian) takes the order.

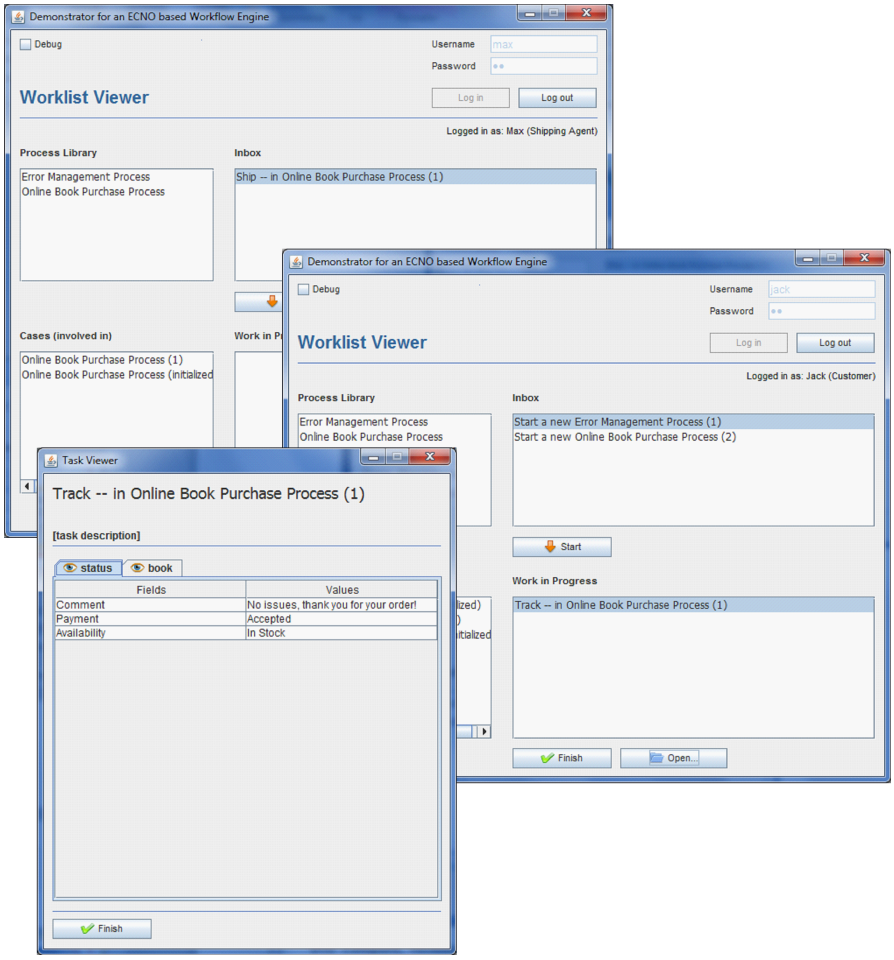

Figure 7.7: Scenario 1: Max (Shipping Agent) ships the order.

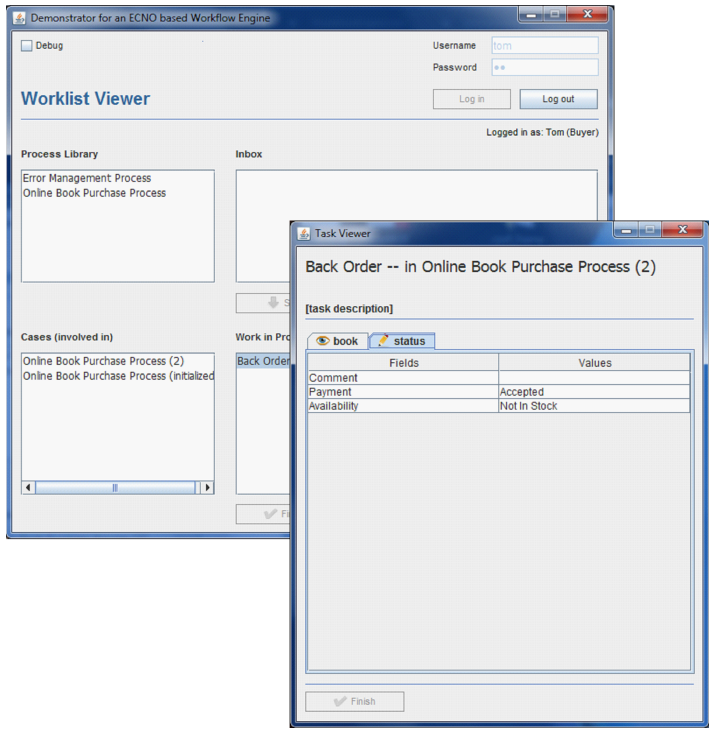

Figure 7.8: Scenario 2: Tom (Buyer) handles an book whish is not in stock.

7.4 GUI testing

The scenario testing on the previous pages showed that the overall functionally was according to the expectations. We will end this chapter with test cases which address specific features of the GUI. The testing has been split into different categories and summarized in three tables.

7.4.1 Test results

Please refer to the Fig. 7.9 containing login and logout related tests, Fig. 7.10 containing tests in the Worklist Viewer and Fig. 7.11 where the Task viewer is tested. Note that these test cases do not refer to specific scenarios like before, but rather defines how the GUI should respond in typical situations - e.g. proper and timely updating. However, we did use Example 2 when performing these tests.

7.4.2 Conclusion

The GUI responds correctly to all the kinds of situations we could think of. The most part of the situations are covered by the test cases in the tables. However there are minor GUI features that are not having test cases. For example, an activity cannot be opened twice, instead the already opened Task Viewer is getting the focus. Another detail is that frames keep focus when they should - for instance, when other frames need to update in the background in response to an action.

#	Test Case	Action and Expected Result	Actual Result	Verdict
1	Initial state of Worklist Viewer.	When the Worklist Viewer starts, only the process view is populated and the only enabled UI elements are the text fields for writing username and password.	(as expected)	Pass √
2	Login (one session).	While the supplied username an password matches those of an agent, the login button enables. When the user clicks on the login button all views (cases, inbox, work in progress) updates and the login button disables. The logout button enables.	(as expected)	Pass √
3	Login (two sessions)	While supplied username and password are matching those of an agent, in two separate sessions, the login buttons enables in both sessions. When one of the users press login, with above mentioned (#2) effect, the login button disables in the other session (the same agent cannot log in twice).	(as expected)	Pass √
4	Logout (one session)	When an agent is logged in and press the logout button, the Worklist Viewer returns to the initial state.	(as expected)	Pass √
5	Logout (two session)	When a logout happens according to #4, and the other session has the username and password entered for the particular agent, the login button enables in the other session.	(as expected)	Pass √

Figure 7.9: Acceptance test: Using Login and Logout.

#	Test Case	Expected Result	Actual Result	Verdict
6	Starting an activity.	The user clicks on an item in the inbox. The Start button activates. When pressing the Start button, the corrosponding item is removed from the inbox list and placed in the work in progress list as a started activity.	(as expected)	Pass √
7	Opening an activity.	When an activity is selected on the work in progress list the Open button enables. When pressed the Task Viewer launches for that particular activity.	(as expected)	Pass √
8	Finishing an activity.	When an activity is selected on the work in progress list, the Finish button shall enables only if the activity can be finished (as defined by the process).	(as expected)	Pass √
9	Starting a case.	If an item in the inbox represents a case that has not yet been started, the text should read "Start a new XX Process..". And when the user clicks on Start, an activity will appear as in #6, in addition a new case will appear in the case list.	(as expected)	Pass √
10	Starting/finishing an activity (two sessions)	If another session is active (another agent logged in) while an action is performed in the first session, and if that action creates a new task or removes an existing task in the inbox if the second session, that effect shall become visible immediately.	(as expected)	Pass √

Figure 7.10: Acceptance test: Using the Worklist Viewer.

#	Test Case	Action and Expected Result	Actual Result	Verdict
11	Show title of task.	The correct title of the activity/task what was opened shall be displayed.	(as expected)	Pass √
12	Showing documents.	Input and output documents shall be displayed in separate tabs, with their process names, and with an icon showing if it is an input or an output document (eye and pen symbols respectively).	(as expected)	Pass √
13	Loading documents.	Documents that exists in the case are displayed with their correct content. When (output) documents do not exist a panel is displayed where the user can create or find a document.	(as expected)	Pass √
14	Creating a document.	If the user clicks on New Document, the panel shall be replaced with a new empty document.	(as expected)	Pass √
15	Finding a document.	If the user clicks on Find, the dialogue Database Browser shall open. When the user has selected a document on the list, the panel shall be replaced with a view of that document with content.	(as expected)	Pass √
16	Editing a document.	In output documents the value column shall be editable but that field name column shall not be. Input documents shall be read only.	(as expected)	Pass √
17	Finishing an activity.	When all output document are created, and when they comply with the optional finish conditions on individual fields, the Finish button shall enable, not before.	(as expected)	Pass √
18	Synchronizing Finish button in Task Viewer with Finish button in Worklist Viewer.	When the Finish button enables in the Task Viewer, the Finish button in the Worklist Viewer shall also enable, provided that that corresponding activity is still selected. If second activity is selected in the Worklist Viewer while the Task Viewer is showing the first activity, they shall independently enable or disable according to the status of the two activities respectively.	(as expected)	Pass √
19	Closing and opening the Task Viewer.	If the user clicks on the "X" in the upper right corner, the Task Viewer shall close. If the activity is re-opened, the content of the documents shall be remembered.	(as expected)	Pass √
20	Finishing an activity from the Worklist Viewer.	If an opened activity is finished from the Worklist Viewer the related Task Viewer shall close.	(as expected)	Pass √

Figure 7.11: Acceptance test: Using the Task Viewer.

Chapter 8

Evaluation

In this section, we will evaluate the results of our case study - the workflow engine. On that basis, we evaluate ECNO in its current state. We assess the results in relation to the goals of the thesis and suggest topics for future work. At times, we will mention when topics from our evaluation could be relevant input to the development of a methodology for ECNO. The section will end with a performance evaluation.

8.1 Evaluation of Workflow Engine

A goal - and a method - of this thesis was to implement an almost realistic workflow engine in ECNO, to gain experience on the process of making applications in ECNO and hereby provide input for the development of ECNO.

What was delivered is a complete and fully functional workflow engine with a presentable a convincing enactment GUI, and with a working process definition tool which integrates seamlessly with the engine. Specifically:

- The solution has reached a high level of completeness in that it covers all three main aspects of business process modelling.

- The architecture maintains the objective of AMFIBIA that aspects can be modelled independently with no bias towards any aspect.
- The architecture also makes it easy to extend the system with new aspects, or to add features to existing aspects.
- For the most part, the solution is implemented in the style that exploits ECNO. For instance, buttons are enabled when corresponding events are enabled, lists displays items representing elements or interactions, and all major functionality in the system is governed by the ECNO engine.
- This thesis, includes two example business processes, where one have been used to develop the workflow engine an the other to test it. The engine can execute multiple process instances in parallel of any type, and the same agents can engage in multiple case at the time.
- The engine can execute relatively complex process model constructs such as concurrent flow, conditional flow (based on data) and execution of control loops.
- The GUI allows agents to log in and out of a Worklist Viewer, even with multiple sessions/users in parallel, to see the task which concerns them. The GUIs update automatically when other users change the state of the process execution. A Task Viewer, which presents the custom defined documents of a task, allows the user to view and edit the documents.
- Acceptance testing has shown that the system executes processes correctly.
- The performance of critical parts of the system have been analysed and optimized (to the extend ECNO version 0.3.1 allows).

As a whole, it meets, if not supersedes, our own expectations of creating an almost realistic workflow engine. In that process, many observations related to ECNO, at the conceptual and technical levels, have been made. We will summarize the observations about ECNO in the following section.

We cannot claim on the basis of this thesis, that a real workflow engine can be made with ECNO in the future but since we came quite far with this attempt, we have added significantly to the evidence, that it might be possible when ECNO has matured further. Further, we have contributed with concrete input to how ECNO can be used in a large application and provided a test platform where future ECNO improvements can be tested and evaluated in an application setting.

To fit this project within the workload of 35 ECTS points a number of compromises were made. Some compromises were due to lack of features (at the right

time) in the ECNO prototype engine. The main limitations of this workflow engine are:

- We create new output documents from the Task Viewer GUI. There is currently no explicit ECNO event which captures this behaviour. It would have been a better design, if we had made an event type CreateDocument in activities. The main reason is, that the creation of documents is what we could call a domain concept (behaviour) of a workflow engine, and therefore it should be made visible in the models.

- Lack of a filtering option in the Inbox and Work In Progress. In a realistic deployment there will be a huge amount of case data in the system, and only the wanted data should be shown in the GUI.

- In the information models, we can only refer to documents that are local to a given process. External documents cannot be referred to. However, at runtime, the user can manually find external documents and use them, although the searching options are very limited. Our information aspect is prepared to be extended with document descriptors, which can refer to external documents.

- The ECNO concepts of event inheritance and event extension have not been used (not available soon enough).

- An interface to a real database is missing. We do support persistency but this solution is based on serializing the EMF resources, which contains our runtime models when the user requests it. Therefore, we cannot guarantee durability (in a ACID sense), because transactions (interactions) can execute, but the new state is not saved until the user requests it specifically.

- Tasks are not executing instantaneously from the point of view of other users. The reason is we allow document objects to be edited directly. This is only a problem if two or more tasks are can use the same document at the same time. In the ACID sense, this violates the isolation property because the concurrent execution of another task writing the the same documents could change the outcome of a task. If business process models do not give write access in two tasks which could execute concurrently the problem is reduced.

- An interface to invoke external applications is missing. Please refer to the WfMC model in Fig. 2.1.

- In the process definition tool, aspect models cannot be placed in separate files.

- The GUI of the process definition tool is quite rough and lacks model validation.
- The organisation aspect has limited support for modelling of what agents could be assigned to tasks at runtime. AMFIBIA suggests a resource descriptor, we have only implemented a very simple version of it (see Sect. 4.6). In the future, this could be replaced by a DSL where more complex rules for assignment could be defined.
- ECNO had limited support for splitting the project into packages and resources in the version we used (ECNO 0.3.1) . With ECNO 0.3.2 (focus on integration) it will be possible to separate parts of the system into packages (for example the aspect meta-models)

None of these limitations are of importance in relation to the objectives of this thesis, and for the most part they are perfectly in line with the expectations to this project that was set from the beginning (see Sect. 1.2).

8.1.0.1 Suggestions for future work

It would be possible to continue with the evaluation of ECNO, still using the case study as method, by creating the next iteration of the workflow engine. This time, the goal could be to get closer to industrial quality. The solution could benefit from the work of this thesis, and from an improved (partly with help of this thesis) ECNO release.

Smaller improvement projects might consider implementing the CreateDocument (and maybe FindDocument too) event type as discussed above. Interactions of this type could be triggered in activities from the Task Viewer. We will leave the analysis of which element types to coordinate with open. However, we could mention that in some tasks the information model might say that document are not allowed to be created at all, only imported from the database. For instance, in our book purchase process the customer should perhaps only have the right pick an existing book and add it as output documents, not to create new books that the store does not have in their inventory.

Another project proposal would be to split the meta-model into different packages in the development workspace to allow several developers to work independently on (and extend) the aspect meta-models and the core meta-model. This would go well together with implementing a more flexible mechanism for plugging the aspect meta-models into the core model. This proposal should not be confused with another option, that of adding support for placing aspects of

business process models (in the runtime-workspace) in different resources, the latter would mainly require additional container classes in the meta-model as well as consideration of the topic of cross resource containments.

Larger projects could look at adding database integration, interfaces to external applications and support for distributed deployment.

Another interesting continuation would be to perform a case study for a completely different kind of application.

8.2 Evaluation of ECNO

There are many implicit ECNO evaluation points in this theses. In this section we will make the main ones more explicit. Distinctions will be made between issues that are conceptional, technical and usability related. These are grouped in individual subsections, even there can be overlap at times. Another important distinction is that of engine issues vs. Eclipse tool issues, but it should be obvious in most cases which category a comment refers to.

8.2.0.2 Overall impression

The overall impression is positive because because the basic concepts of ECNO covered most of the needs for implementing the workflow engine and seemed to apply quite naturally in most cases. Actually we achieved very simple and elegant modelled solutions to complex problems (for instance, integration of aspects and realization of the control aspect with modelled Petri net semantics).

One of the purposes of ECNO is to reduce the distance from analysis to implementation, and this appears to be very achievable with ECNO. Having a clear domain model which includes the right events and coordinations is was definitely most of the work (ignoring here the lack of an interactions debugger) on this particular workflow engine implementation, and the models never became obsolete, as they became the source. The code snippets that we used were also part of the source, but they were very trivial to program.

The fact that code snippets written in the underlying programming language (in this case Java) are easily integrated with the models, means that as a developer you are never stuck due to a "missing" ECNO modelling feature, and are never forced to model behaviour that really should be coded (for example algorithms).

There is another side to this coin. It also means that ECNO gives the developer the freedom to define events for functionality, or to just program it from an integrated code snippet. We spent a great deal of time considering when events should be defined, and for what kind of behaviour, and when they would be overkill, or just not appropriate. We expect other newcomers to ECNO would face the same issues. We could probably relate this to the fact that no books or papers have been written about methodologies in ECNO development (actually a methodology report is planned by Kindler). For the behaviours that are cross cutting, the choice is easy, here events are obviously appropriate. For simple value getters, lookups and factory operations, it's usually an easy choice not to define events. In the space between these examples the choice is often less easy.

8.2.0.3 Conceptual issues

In the early phase of this thesis it was found that when using event inheritance, where one event inherits the behaviour of another event, the event parameters defined in the parent event were not available in the derived event. This issue was fixed in ECNO in parallel with this thesis, but we have not used the new feature.

Later on, the need for using additional parameters within specific aspects for certain event types was identified. The best example is the parameter state in the control aspect in StartActivity, which lead to the introduction of StartActivityC as a workaround. Recall, that we didn't want the parameter state to be known in the core or in other aspects (separation of concerns). ECNO has now been updated to support event extension, a feature which allows derived events to add parameters to the events, but we did not have time to update our models.

Debugging interactions is very difficult with ECNO right now (version 0.3.1), and probably the most time consuming part of developing with ECNO. In particular, when interactions which you expect to be found (calculated) by the engine are not found. There is no easy way to extract from the engine which part of the model prevents the interactions. We have tried in this thesis to add a logging feature to the interaction algorithm but without much luck - e.g. the logging worked but the reason for the error was not identified. This issue is conceptual since it it not trivial to define how a debugger should work. How do you inform the debugger what you expected (and when), and in which format does it report back what really happened? We found an effective but still time consuming debugging method for debugging interactions without a debugger: The typical situation is that an interaction is missing at some stage in the execution (the developer expected it was there). Let's say that 5 to 10 elements

are linked with coordinations for this event type and they each have local conditions. ECNO offers no indication of which elements is preventing a possible interaction and identifying that or those elements can be like finding a needle in the hay. A systematic approach to debug this, is to first remove all coordinations from the global behaviour model and regenerate the software. If there is still no interaction, we known the trigger element is responsible. From this point on, coordinations can be added one at a time until the blocking element is found. Perhaps our debugging trick can be a supported feature (among more) of a future debugger. This point is as well related to methodology.

We had a problem when we wanted several participants of an interaction to cooperatively build a data structure with one or more new element(s) in them, in particular, building structures of activities referring their activity aspects. During execution of the respective actions of the participants, information cannot be shared. Refer to the discussion in Sect. 4.7.3. Kindler suggested that perhaps actions could be divided into stages in a future ECNO release, such that new object could be created in the first stage, and then shared before the second stages would execute (note, this was just an initial thought).

When an element is added to the engine, all registered controllers are notified. It is not possible to for a controller to specify which element types it cares about, or don't care about.

8.2.0.4 Usability (for developers)

The modelling tools and the code generator are all reliable. The code generator is easy to use but the modelling tools lacks some features, especially model and syntax validation, and their usability could generally be improved.

Navigating to the local behaviour of element types, requires that you open another file then locate the element type on a long list, and then open the behaviour diagram (which by the way does not show the element type name when opened). It would save a great deal of time if you could navigate open a local behaviour diagram by double clicking (or similar) on an element type in the global behaviour diagram.

As it is, the Ecore model and the ECNO global behaviour diagrams are edited in two different tools (the ECNO model is referring to the Ecore model). Although the two modes would have to exist as separate resources, it would be nice to have a combined editor. This point relates to methodology.

8.2.0.5 Technical issues

This is a list of technical issues that were identified while working on this thesis: We also list issues which were resolved in ECNO (by Ekkart Kindler) during this thesis. Some readers might prefer to skip this section.

- The global behaviour editor allowed links between elements which did not have references between them in the Ecore model. It had the consequence that all references had to have unique names in the Ecore model to be able to locate the right one in the ECNO mode *(status: Resolved in version 0.3.2)*.
- ECNO projects were limited to a single Ecore package *(status: Resolved)*.
- Adding and getting event parameters from controllers were quite complicated *(status: Partly Resolved)*. A small modification in ECNO combined with ECNO Connectors (see Sect. 6) this problem is now more manageable.
- In local behaviour diagrams event parameters are referred to by position in the event parameter list, see any local behaviour diagram in this report. It can lead to errors - and has done so in this project. *(status: Partly Resolved)*. With the latests ECNO release, parameters are referred by name.
- ECNO global behaviour diagram gets corrupted when creating a reference before defining the type of target element *(status: Unknown)*.
- Element with coordination link to "null" appears to cause an unrecoverable exception *(status: Identified)*.
- A syntax error in action label of transitions in local behaviour has the effect that no code is generated. Therefore the error does not show itself in the generated code *(status: Resolved in version 0.3.2)*.

8.3 Performance evaluation

In the previous work on ECNO itself, performance has not yet been a priority yet while the focus has been on the concepts so far, hence it has not been a priority in this thesis. Actually, if the an objective of this workflow engine had been great performance, we would probably have made some other design choices along the way. However, for the future, performance is an important

topic in ENCO. Therefore, an initial performance analysis with the workflow engine was made to provide concrete information for future work to build on. The findings are presented here.

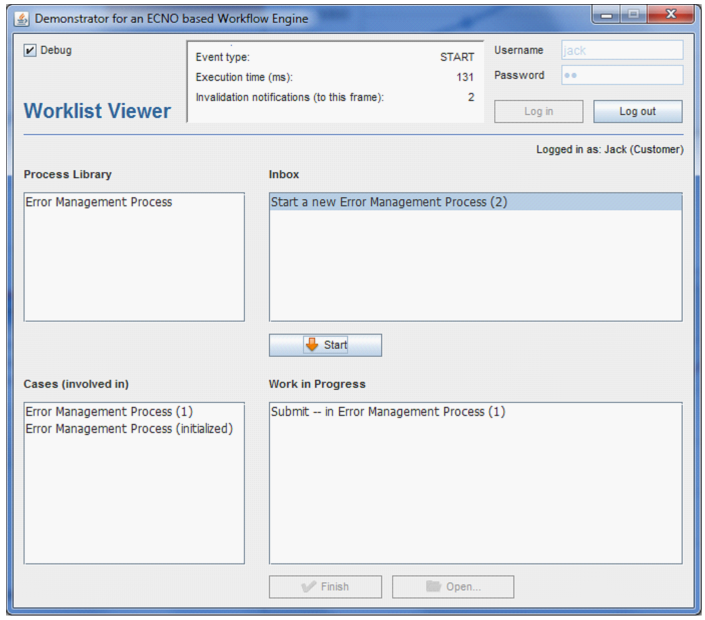

Figure 8.1: Debug panel.

We added a debug panel to the Worklist Viewer (Fig. 8.1). For now, it displays the execution time of the latest executed interaction as well as the number of invalidation notifications received by the GUI instance. The debug panel has the limitation that notifications sent to other GUI instances are not counted, even they can block the current GUI instance too.

8.3.1 Performance of Start and Finish vs. number of cases

The performance was tested vs. the total number of created cases. This was done with a single user session - only one agent is logged in. We focussed on the events for starting and finishing activities. Please refer to the test results in Fig. 8.2.

The curves shows that the execution time for *start* is proportional to the number of cases, while the execution time for *finish* is independent on the number of

cases. Recall that in the performance discussion in the Sect. 5.6.2 the aim was to only update the inbox list with respect to interactions in affected cases. It appears from the graph that this idea is working for *finish* but not for *start*.

Looking more carefully at this, it was found the GUI is working correctly, but when an activity is started the interactions iterators, that we registered the GUI with, are sending invalidation notifications (for all cases). In contrast, when an activity is finished, the GUI receives only one notification, corresponding the particular case we finish the activity in. So, why does this happen?

When starting an activity, we modify the agent element (by adding a reference to the activity). Since the agent was involved when ECNO calculated the interactions in each of the cases, ECNO is forced to invalidate them, because the engine does not know what we know. In particular, that in our design, starting a new activity, and referencing it from an agent, cannot affect which interactions are possible in other cases. We can conclude everything is working at it should, but there is still a critical performance issue when starting activities.

One way to avoid this problem in the application, would be to avoid setting the reference from agent to activity. However, the reference is there for a reason, so that would be a bad idea creating other problems. Most likely this issue should be solved by adding additional functionality to ECNO. For example, one could imagine the application to configure that certain references should not trigger invalidations. It is outside the scope of this thesis to analyse the problem further, however the workflow engine we implementation should be a good validation platform for related features in ECNO in the future.

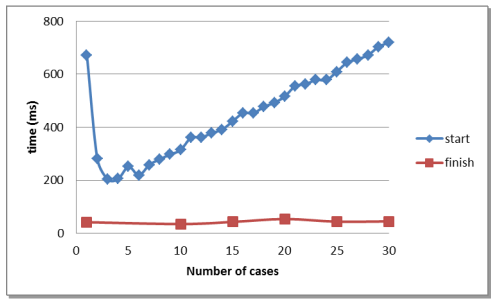

Figure 8.2: Start and Finish - as function of the number of cases (one session).

8.3.2 Performance of Finish vs. number of sessions

The performance of *finish* was tested vs. the number of users logged into the system. We omitted *start* because we identified another problem already which make it harder to conclude on the effect of the sessions. The number of cases was kept constant (10). Please refer to the rest results in Fig. 8.3.

The curve looks polynomial to begin with but tends to be linear over the last four measurements. We would expect a linear curve here, because we're letting the same CPU handle updates (of the interaction in a single case) in all sessions synchronously. In a real distributed workflow engine deployment the expectation would be much different. We will not analyse this aspect here, nor the exact reason for the polynomial beginning of the curve.

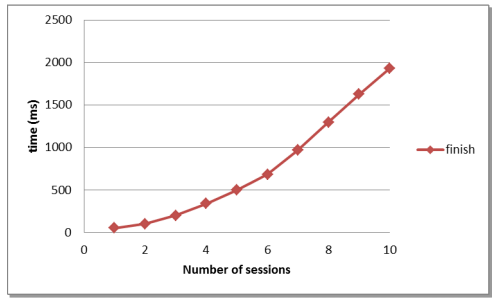

Figure 8.3: Finish - as function of the number of user sessions.

8.3.3 Performance of Login vs. number of sessions

The performance of *login* was tested vs. the number of other users logged in at the time. The number of cases was kept constant (10). Please refer to Fig. 8.4 to see the test result. The curve shows a polynomial relationship. We had hoped the login time would be unaffected by the number of other sessions.

We found a major part of the explanation is due to the same kind of problem as was identified earlier. When an agent logs in, all other session receive invalidation notifications, beucase they listen to interactions calculated in the same cases, the cases refers to all involved agents, and one agent has changed by logging in. There is no need to update, logically speaking, since the resulting interactions are the same - however, ECNO cannot know this. This would explain at a linear curve, but why is it polynomial?

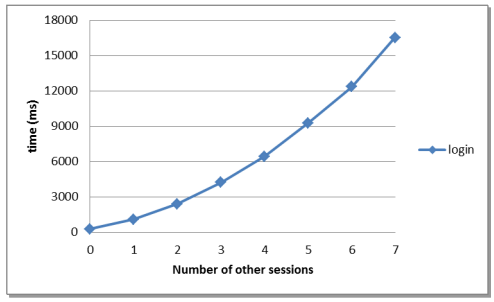

Figure 8.4: Login - as function of the number of user sessions (10 cases).

Actually, there is another factor (besides the number sessions to update) which increases to, when an agent logs in. As figure shown in Fig. 8.5, the time it takes to update one session in the background is increasing. Notice the time spent to update Ellen's view (when Jack logs in) is increasing as a function of sessions.

So, based on where we have inserted the timer in the code, we can conclude that ECNO is spending increasing time to evaluate an interaction "for Ellen", when additional agents are logged in. The likely explanation is, that the possibilities, which ECNO must evaluate, increases, becuase logged in agents can do more than logged out agents. Refer to the local behavior of an agent in Fig. 4.26.

Other sessions	Login time (Jack)	Update time (Ellen)
0	~ 100 ms	n/a
1	~ 200 ms	~ 70 ms
2	~ 380 ms	~ 120 ms

Figure 8.5: Login - update time for session in the background (1 case).

8.4 Summary

In this section we evaluated our workflow engine and then ECNO. We also performance an initial performance analysis. In the following section we conclude on this thesis.

Chapter 9

Conclusion

The goal of this thesis was to realize a workflow engine using the Event Coordination Notation (ECNO). The workflow engine was to follow the ideas in the meta-model for business processes known as AMFIBIA. We have achieved a working workflow engine, which has, at least, the level of realism, which was expected from a thesis of this size. We allowed from the beginning to omit certain features, since the main purpose of realising a workflow engine was to evaluate the ECNO, e.g. not to create a workflow engine for end users.

We have realized a fully working workflow engine with ECNO, in the spirit of AMFIBA. The workflow engine comes with an enactment GUI to demonstrate that it works. The created software is able to log in several enactment users at the same time, and execute multiple business processes in parallel. The processes, which the workflow engine can execute, include the main aspects of business processes: organisation, information and control. We also developed a simple process definition tool based on EMF. This tool integrates seamlessly with the workflow engine.

This report includes a presentation our our work (the workflow engine), an evaluation of this work and then of ECNO. Conceptual and technical feedback has been noted and documented in this report. We have additionally provided input for a methodology for ECNO, as well as input for performance topics.

Regarding the strengths of ECNO, we will mention the main ones here. We found that the integration with UML domain models works well, and also in practise ECNO lives up to the intention of narrowing the gap between domain models and implementation models. We saw that the ECNO concepts, and the current tools, were now mature enough for generating a fully working application. In addition, ECNO allows separating generated code from other code (manually coded), which makes applications quite easy to maintain - e.g. it is not a big issue to update the models and re-generate the code (see Sect. 6.4.1).

With regards to the main limitations, we demonstrated that ECNO's notification system for invalidation of interactions (possible behaviour at a given time) still needs some additional concepts that will allow application developers to optimize the performance. Our application was slowed down significantly by too many redundant updates that we could not prevent. Another limitation is that ECNO does not come with a debugger for interactions, making it quite painful to find interaction related errors. We found a systematic, but slow, method for debugging interactions, which could possibly be supported (made easier) by a future debugger (see Sect. 8.2.0.3).

Finally, we will conclude, that the results of this project have added to the evidence that with continued improvements, ECNO might be able to support the development of industrial quality applications in the future.

Bibliography

[1] Kindler, E. *Integrating behavior in software models: And event coordination notation - concepts and prototype.* In: Third Workshop on Behavioural Modelling - Foundations and Applications (BM-2011), Proceedings. (2011)

[2] Axenath, B., Kindler, E., Rubin, V. *AMFIBIA: A Meta-Model for the Integration of Business Process Modelling Aspects.* (2005)

[3] Axenath, B., Kindler, E., Rubin, V *The Aspects of Business Processes: An Open and Formalism Independent Ontology* (2005)

[4] Harrison W, Ossher H. *Subject-Oriented Programming (A Critique of Pure Objects)*

[5] Sommerville, I. *Software Engineering rev. 9.*

[6] Van der Aalst, W., Van Hee, K. *Workflow Management – Models, Methods and Systems* (2000).

[7] Leymann, F., Roller D. *Production Workflow: Concepts and Techniques* (1999).

[8] Kiczales G., et al. *Aspect-Oriented Programming* (1997).

[9] C. A. R., Hoare *Communicating Sequential Processes (CSP)* (1985).

[10] Hollingsworth *Workflow Management Coalition, The Workflow Reference Model* Issue 1.1 (1995)

[11] Kindler, E. *An ECNO semantics for Petri nets* (2012)

[12] Mathias, W. *Business Process Modelling* (2007)

[13] Kindler, E. *Coordinating Interactions: The Event Coordination Notation* (2013) Unofficial Draft Version of a tehnical report on ECNO

[14] Kindler, E. *Model-based software engineering: the challenges of modelling behaviour* (2010)

[15] Selic, B. *The Pragmatics of Model-Driven Development* IEEE Xplore (2008)

[16] France, R., Rumpe, B. *Model-driven Development of Complex Software: A Research Roadmap* (2007)