Thesis to obtain the Diploma Degree at the TU Berlin

Development of a Visual Service Design Tool providing a mapping from BPMN to JIAC

Tobias Küster
tobias.kuester@dai-labor.de

April 18, 2007

DAI Laboratory
Technical University of Berlin
Faculty IV – Electrical Engineering and Computer Science
Prof. Dr. Sahin Albayrak
Die selbstständige und eigenhändige Anfertigung dieser Diplomarbeit versichere ich an Eides statt.

Berlin, ____________________________________________

Tobias Küster
Abstract

Although they have been topic to academic research for several years and although they are often seen as the next step towards truly reusable and modular programs, multi-agent systems are still underrepresented in the industry. At the same time web-services and service oriented architectures are adopted much faster in the same domain. A possible reason for this can be found in the lack of easy-to-use tools for the model driven creation of multi-agent systems.

In this thesis a tool for the model driven development of programs for the multi-agent system JIAC will be presented. For this purpose a mapping from business process diagrams based on the Business Process Modeling Notation to JIAC IV has been developed. Based on the Eclipse framework a visual editor for the creation of BPMN diagrams has been implemented. Diagrams created with this editor can be validated and translated to executable JIAC programs using a rule based transformation tool.

Zusammenfassung


Acknowledgment

I want to thank Axel Heßler and Holger Endert for their support and suggestions. Further I want to thank the entire DAI Laboratory for providing me with a comfortable working environment and an interesting and challenging topic for my thesis.

Special thanks go to the TFS Group for providing us with the latest version of the EMT framework and to the Eclipse GMF newsgroup for answering all of my questions.

Finally I want to thank my whole family for their constant support.
### Contents

1 Introduction .................................................. 1
   1.1 Motivation and Problem Description ..................... 1
       1.1.1 Model Driven Engineering ............................ 2
       1.1.2 Process Modeling and Business Process Diagrams ....... 2
   1.2 Goals .................................................... 3
   1.3 Outline .................................................. 4

2 Process Modeling Notations and Agents .......................... 5
   2.1 Introducing JIAC .......................................... 5
   2.2 A Survey of Process Modeling Notations ................. 7
       2.2.1 Petri Nets ........................................... 7
       2.2.2 UML Activity Diagrams ............................... 8
       2.2.3 Business Process Modeling Notation .................. 9
       2.2.4 XPDL .................................................. 10
       2.2.5 Simple Notation ..................................... 10
       2.2.6 Proprietary Notations ............................... 11
   2.3 Comparing BPMN and UML Activity Diagrams ............ 11
   2.4 A Closer Look at BPMN ................................... 12
       2.4.1 BPMN Elements ....................................... 12
       2.4.2 Levels of Complexity ................................ 15
       2.4.3 Export and Code Generation ......................... 16

3 Model Driven Engineering ...................................... 17
   3.1 Using MDE for Designing Eclipse Editors ................. 18
       3.1.1 The Eclipse Modeling Framework ...................... 19
       3.1.2 The Graphical Modeling Framework ................. 21
| 3.2 | Using MDE for Modeling Processes | 23 |
| 3.2.1 | Unstructured Workflows | 23 |
| 3.2.2 | From BPMN to BPEL | 26 |
| 3.2.3 | From BPMN to Agents | 27 |

| 4 | Model Transformation Tools | 29 |
| 4.1 | Eclipse Generative Modeling Tools | 29 |
| 4.2 | The Tiger EMF Transformation Project | 30 |
| 4.2.1 | Transformation Rules | 30 |
| 4.2.2 | Backtracking and Queries | 31 |
| 4.2.3 | AGG and EMT | 31 |
| 4.2.4 | Interpretation and Compilation of Rules | 32 |

| 5 | The Mappings to BPEL and JIAC IV | 33 |
| 5.1 | BPEL | 33 |
| 5.2 | JIAC | 33 |
| 5.2.1 | Modeling Agents with Custom Artifacts | 34 |
| 5.2.2 | Element Mapping | 34 |

| 6 | Implementation: The Editor | 41 |
| 6.1 | Project Structure | 41 |
| 6.2 | The Domain Model | 43 |
| 6.2.1 | Containment | 43 |
| 6.2.2 | Attributes | 44 |
| 6.3 | The Editor Specification | 45 |
| 6.3.1 | Customizations | 46 |
| 6.3.2 | Validation | 48 |
| 6.4 | Open Issues | 48 |
| 6.4.1 | Attribute Sets | 49 |
| 6.4.2 | Expressions | 50 |
| 6.4.3 | Tasks and Subprocesses | 50 |
# List of Figures

1.1 Sample BPMN Diagram ......................................... 3
1.2 Our Vision of the Visual Service Design Tool ............... 3

2.1 The JIAC metamodel ........................................... 6
2.2 Use of one PML throughout software engineering ........... 7
2.3 A simple producer-consumer petri net ......................... 8
2.4 E-mail example as UML Activity Diagram .................... 9
2.5 E-mail example in Business Process Modeling Notation .... 9
2.6 E-mail example in Simple Notation .......................... 11
2.7 BPMN Event types ............................................ 13
2.8 BPMN Event sub types ....................................... 13
2.9 BPMN Activity types ......................................... 13
2.10 BPMN Gateway types ........................................ 14
2.11 BPMN Connection types ..................................... 14
2.12 BPMN Swimlanes ............................................. 15
2.13 BPMN Artifacts .............................................. 15

3.1 Model driven software development lifecycle ................ 18
3.2 Code generation with EMF .................................. 19
3.3 Ecore metamodel ............................................. 20
3.4 The GMF Dashboard .......................................... 21
3.5 Basic Structured Workflow Models ......................... 24

4.1 EMT Visual Rule Editor ...................................... 30

5.1 Custom Artifacts ............................................. 34

6.1 Editor-Plugin Interdependencies ............................ 42
1. Introduction

In this first chapter we want to introduce the reader to the topic to be covered in this thesis. We will state the problem and explain the motivation for this work and the relevance it has in the field of agent oriented software development. Finally we will describe how we attempt to solve it and give an outline of the following chapters.

1.1 Motivation and Problem Description

In the past years and decades multi-agent systems have evolved from science fiction to a promising field of research. In fact they could mark the next great leap in software architecture after imperative and object oriented programming by bringing the principles of reusability and modularity to a new level.

<table>
<thead>
<tr>
<th>Imperative Programming</th>
<th>Programs can be subdivided into several functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Oriented Program-</td>
<td>Objects encapsulate data and functionality and can be shared between programs of the same language.</td>
</tr>
<tr>
<td>ming</td>
<td></td>
</tr>
<tr>
<td>Multi-Agent Systems</td>
<td>Functionality is provided by agents and can be shared among programs of different languages. Services can be selected autonomously based on their precondition and effect.</td>
</tr>
</tbody>
</table>

Software agents and multi-agent systems have been researched for a long time now. But although they provide advanced concepts, some of which are now rediscovered by Service Oriented Architecture and web-services, they are still mainly an academic subject, while the uptake of agent technologies by the industry is very slow. Although they address the same problem, web-services have been adopted a lot faster and are by now well established in the business world.

A reason for this could be the complexity of multi-agent system that might discourage business people and even experienced developers from giving agents a chance. The mentalistic notions used in agent technology seem to be too unfamiliar to the business people, who rather tend to think in terms of processes and web-services.

To bridge that gap that exists between the academic research and the industry and to facilitate the creation of multi-agent systems our approach is to allow agents to
be designed and deployed using business process diagrams and thus to widen the clientèle of agent technology.

1.1.1 Model Driven Engineering

Following the principle of Model Driven Engineering programs are no longer written, but specified in a number of models. These models are then used for generating the actual program code, facilitating and accelerating the software development process and resulting in high quality software. The modeler does not have to care too much about a language’s details and the resulting programs are uniformly designed and thus easy to maintain and highly interoperable.

- Usage of several models for different views on a system, for instance for control flow, data structures and distribution
- Independence of platform and programming language: One model can be used to generate the same programs for various languages and platforms.
- Automation of laborious tasks such as GUI building, exception handling and resource management
- Understandable, easily maintainable code
- Model validation

With the right metamodel and a mapping from that metamodel to agent concepts, Model Driven Engineering could be a great help when designing multi-agent systems.

Further, given today’s model driven development tools, also feature-rich, visual editors for any domain can be designed and generated in a short time. Being created for designing business process models these editors then again can be used for the generation of executable programs based on these processes.

1.1.2 Process Modeling and Business Process Diagrams

Business Process Diagrams provide a very abstract view on a program. Although UML covers more aspects of software development and might be more popular among the developers, business process diagrams can quickly be drawn and are easily understandable for managers, business people and domain experts who are not always familiar with UML.

The Business Process Modeling Notation (BPMN) is a relatively new process modeling notation. As a standardized business process notation it is increasingly important. At the same time the clear and concise graphical notion, as seen in figure [1], can be enriched with non-graphical elements and attributes and extended with artifacts, enabling a mapping to executable programming languages.

However, the potential of mapping BPMN to executable languages suffers from the problem that BPMN diagrams can be arbitrary directed graphs while most programming languages are block structured, and thus BPMN has more expressive power than most programming languages. This issue has been discussed in many
1.2 Goals

Our goal is to develop a Visual Service Design Tool (fig. 1.2) that can be used for modeling business process diagrams and transforming these diagrams to executable JIAC programs. As a byproduct also a transformation to BPEL will be provided.

![Figure 1.1: Sample BPMN Diagram](image)

![Figure 1.2: Our Vision of the Visual Service Design Tool](image)

The main challenges and requirements for achieving this are:

1. The creation of the business process metamodel to be used for the process diagrams’ internal representation. The model has to conform to a standardized business process notation so it can be intuitively understood by business people, but still has to be to some amount extensible. The model has to provide enough semantics to enable a mapping to executable languages, but still has to be easy to create and understand.
The development of an easy to use, yet powerful editor for the creation of business process diagrams. Since the aim of this work is to facilitate the creation and to boost the distribution of multi-agent systems the editor has to be easy to use so it can be operated by non-experts, business users and domain experts, too. It has to provide the usual editing features and a way to validate the diagrams to conform to the underlying specification.

The development of a mapping from BPMN to JIAC IV. While most elements of BPMN map nicely to various JADL statements the original BPMN notation lacks an equivalent to JIAC Agents, thus we will have to discuss whether one of the existing BPMN elements can be used for this or whether the specification will need to be extended by additional elements for this purpose.

The implementation of that mapping as a transformation tool. The export features have to be extensible and usable together with the editor. The structural differences between BPMN diagrams and BPEL and JIAC programs will require a number of rules for identifying and transforming block structured in BPMN diagrams.

With this tool the user can draw a high-level BPMN process diagram and generate code from it. Depending on the level of detail of the process model the resulting program will still need some refinement, but given a very detailed model the program could be executable right after its creation.

1.3 Outline

In chapter 2 we will introduce the reader to the agent description language JADL and the JIAC framework and give an overview on existing process modeling languages that might be adequate for modeling agent systems. We will explain why we decided for BPMN and list its most important features.

In chapter 3 we will give a short introduction to Model Driven Engineering and have a look on EMF and GMF. Further we will discuss the model driven development of processes, especially when based on unstructured process models, and list some existing approaches in transforming business process diagrams to executable BPEL processes and agent systems.

After that, chapter 4 will introduce the rule-based transformation framework being used in this work.

In chapter 5 the mappings from BPMN to BPEL and JIAC will be explained in detail.

In chapter 6 we will review the development of the Visual Service Design Tool and in chapter 7 we will describe the transformation modules used for the mapping from BPMN to BPEL and JIAC. We will outline the concept, have a look on the several stages of the mapping and introduce the most important rules. Chapter 8 will give some examples of the transformation from BPMN to JIAC.

Finally, chapter 9 will give the conclusion of this thesis and state some ideas and topics for future work to be done on this topic.

In the appendix of this work the reader can find a user manual for the diagram editor.
2. Process Modeling Notations and Agents

A possible reason for the slow uptake of agent concepts in industry is the gap that exists between the mentalistic notions used in agent theory and the business people’s mindsets of services and processes. The business world has been occupied by processes and web services. Thus the best way to establish agent technology and to close that gap seems to be providing a method to translate business process diagrams to agent systems.

The reason we decided for business process diagrams instead of UML is that we want to address the business people, which preferably use business processes, while UML is used mainly by the developers.

In the first section of this chapter we will briefly introduce JIAC, the main target language in the mapping to be described in this work. After that we will have a look on the various existing process modeling notations. We will explain why we picked BPMN as the source language for the transformation, comparing it with UML Activity Diagrams. Finally we will provide an overview on the most important elements and features of BPMN.

2.1 Introducing JIAC

The Java Intelligent Agent Componentware (JIAC) is a framework for agent oriented software development. The Core of JIAC is the JIAC Agent Description Language (JADL) [20]. The JIAC framework provides features such as FIPA compliant communication, Believe-Desire-Intention (BDI) reasoning, strong migration, web-service connectivity and others. Further it provides high security (Common Criteria EAL3, certified by the Federal Office for Information Security of Germany, BSI) and advanced accounting mechanisms, making it suitable for the use in industrial and commercial applications. The JIAC framework comes with a runtime environment and a toolsuite for the creation of agents.

JIAC agents are knowledge-based and consist of dynamic components – Facts, Rules, Goals, Plans and Services – that can be exchanged at runtime (see figure 2.1).
2. Process Modeling Notations and Agents

Facts are instantiations of categories, defined in JADL ontologies, which make up the agents’ shared vocabulary. They represent the agent’s knowledge of the world and himself, his “state of mind” to say to, and are used for instance for reasoning.

Rules and Goals are used to control the agents behavior. Rules make the agent react on changes in its environment, that is, its knowledge base. Using Goals, the agent, following the BDI paradigm, will autonomously select a Plan or Service to make its Goal come true.

Plans and Services define what the agent can do. For their description JADL is used. Services are made up of Plans and normally have either a user and a provider side or only a provider. More sophisticated patterns, such as contract net, are possible, too. For the communication between the user and provider roles Speechacts are used. In most cases the planelements consist of a script, which is basically similar to other programs with the usual control-flow constructs, such as branching, loops, sequential, alternative and parallel execution. However, JADL provides a number of rare features, like using unification for both the evaluation of formulas and for making assignments to variables. JADL is based on a three valued predicate logic (with possible values true, false and unknown) suitable for open-world problems in uncertain environments. Services and Plans are selected depending on their preconditions and effects.

Finally JIAC agents can be supplemented with Java components, so called Agent-Beans. These components allow the agent to interact with the real world, e.g. by providing user interfaces or a robot control.

JIAC already comes with a set of predefined agentroles, which can be aggregated to agents. These agentroles provide functionality for knowledge access, reasoning, mobility, the management of groups of agents as a agent platform and many more.

The mapping which we will describe in this work will focus on plans and services.
2.2 A Survey of Process Modeling Notations

Process Modeling Languages (PMLs) provide a very abstract view on a program or process. The central point of PMLs is the modeling of control flow, including branching and parallelism, communication and activities. Unlike UML, data modeling and data flow are subordinated elements. Using an easily understandable notation PMLs are best suited for communicating process specifications throughout the whole software development process, from the stakeholders and domain experts to the software architects and developers.

While there are some textual PMLs, such as BPEL, most PMLs have a graphical notation, and most of these graphical notations are very similar: Activities, which normally are complex operations such as method or service calls, are represented by nodes, which are connected by arrows, representing the control flow and the exchange of messages.

With today’s Model Driven Engineering the process model can be used throughout all stages of software engineering, from requirements analysis to implementation (see figure 2.2). Of course other models and diagrams will be needed, too, covering those issues that are not topic to process modeling. However, the process model should fit the needs of all parties involved in the software development process: From business users and domain experts over business analysts and architects to developers. The notation has to be simple and intuitively to understand, yet meaningful, formal and non-ambiguous.

![Figure 2.2: Use of one Process Modeling Language throughout all stages of software engineering](image)

Currently there are many different process modeling languages available, some of which are standardized, and even more are proprietary. In this section we will have a look on some common workflow and process modeling notations. We will give a brief description of the language and state its pros and cons with respect to the problem to be solved in this thesis.

2.2.1 Petri Nets

Petri Nets have a very formal, mathematical grounding. Mathematically a Petri Net is a 5-Tuple \( PN = (P, T, F, W, M_0) \) with \( P = \{p_1, \cdots, p_n\} \) (Places), \( T = \{t_1, \cdots, t_m\} \) (Transitions), \( F \subseteq (T \times P) \cup (P \times T) \) (Flow), \( W : F \rightarrow \{1, 2, \cdots \} \) (Weight) and \( M_0 : P \rightarrow \{0, 1, 2, \cdots \} \) (Marking). Visually the Places are represented as cycles, Transitions as bars, Flow as arcs, Weight (if \( > 1 \)) as labels on the arcs and the Marking as Tokens on the Places (see figure 2.3 for a simple example). Beside this definition there are many other types and extensions of Petri Nets, from simple Elemental Petri Nets, where the Weight is 1 for each Flow and Places have a maximum capacity of one Token, to colored, timed or prioritized Petri Nets.

With \( F \subseteq (T \times P) \cup (P \times T) \) a Petri Net is a bipartite graph, meaning that the flow control, the arcs, may only go from a Place to a Transition or vice versa, but
2. Process Modeling Notations and Agents

Figure 2.3: A simple producer-consumer petri net

neither from a Place to a Place nor from a Transition to a Transition. A Transition is active if on each Place before the Transition there are as many Tokens as the arc’s Weight. When an active Transition fires Tokens are moved from the Places before to the Places after the Transition, according to the arcs’ weight. Thus concurrency can be modeled with Places having multiple outgoing arcs, while parallelism is modeled with Transactions having multiple outgoing arcs (analogue for merging and synchronization).

Another particularity of Petri Nets is that they do not only model a process, but also the state the process currently is in, represented by the Marking (the Tokens). For more detailed information about petri nets please refer to [24].

Petri Nets are well-established and used throughout workflow management and software design for analysis and validation of behavioral properties such as Liveness, Reachability and Boundedness, for instance to detect and avoid possible deadlock situations.

But while Petri Nets can be used to model and analyze nearly everything from basic producer-consumer systems to multiprocessor systems their notation is very abstract and hard to understand, not only for non-experts. For the same reason it does not seem to be reasonable to use Petri Nets to generate executable programs from. Thus they would not be a good choice for a tool intended to facilitate the model driven creation of agents using business processes.

2.2.2 UML Activity Diagrams

UML Activity Diagrams is one of the many diagram types used in the Unified Modeling Language (UML) [26, Chapter 12], which clearly is the most popular modeling notation. Since software development is often accompanied by creating UML diagrams it might stand to reason to use UML Activity Diagrams for modeling business processes.

The semantics of Activity Diagrams have changed from UML 1.x to UML2. While in UML 1.x Activity Diagrams were very similar to State Diagrams they have now been enhanced with more possibilities for modeling parallelism and communication via asynchronous communication. However, Activity Diagrams lack the information given in the other UML diagrams, and modeling them all can be difficult and time-consuming, especially for small-scale enterprises.

The core elements used in most UML Activity Diagrams, like boxes, edges and diamonds, are easily to understand for all business users (see figure 2.4 for a simple example). However there are some rarely used, yet important elements which might
be unfamiliar to non-experts, like interruptible regions, expansion regions and parameter pins, providing ways for modeling exception handling and user interaction or iteration of collections of input parameters or object flow and signals.

![E-mail example as UML Activity Diagram](image1)

The elements are very clear and can be drawn easily by hand. But many elements of UML Activity Diagrams look very similar or ambiguous, like activities with parameter pins and expansion regions, which could lead to some confusion among unexperienced modelers.

### 2.2.3 Business Process Modeling Notation

The *Business Process Modeling Notation* was first published by the BPMI and has later been adopted by the OMG. The goal of the development of BPMN was to create a standardized modeling notation for business processes and by that to reduce the confusion created by dozens of proprietary business process notations. The BPMN is intended to "consolidate the best ideas from these divergent notations into a single standard notation" [36, p. 9].

There are four basic categories of element types in the notation: Flow Objects (Events, Activities and Gateways), Connecting Objects (Sequence Flows, Message Flows and Associations), Swimlanes (Pools and Lanes) and Artifacts (Data Objects, Groups and Annotations). Please refer to section 2.4 for more detailed information about the several elements.

![E-mail example in Business Process Modeling Notation](image2)

Figure 2.5 shows a simple example for an e-mail client periodically looking for new mail. The elements are quite self-descriptive and most of them are already known from other notations, so the basics of the BPMN are readily understandable for all business analysts, architects and developers and even for non-experts. At the same time BPMN provides a large variety of subtypes for each of the Flow Objects and every element type is enriched with many non-graphical attributes, making the
models sufficiently detailed for being exported to executable languages while keeping the visual notation concise and understandable.

A problem with BPMN is that it is mainly a notation. Although the specification describes many non-graphical attributes and a mapping to a formal language, it neither states an exchange format, like an XSD, nor clear semantics for all of the elements. Still the Business Process Modeling Notation can be used throughout the whole software engineering lifecycle, from a simplified model at the requirements analysis up to a highly detailed model that can be used for generating code for an executable language.

### 2.2.4 XPDL

The XML Process Definition Language (XPDL) \[35\] is a workflow model interchange language standardized by the Workflow Management Coalition (WfMC). XPDL does not have a graphical notation on its own. Instead the goal of XPDL was to be an interface, a “lingua franca” \[2\], for the various diverging workflow notations and tools. In the beginning XPDL defined only a minimum set of core elements, but later many vendor specific extensions, including a large part of BPMN, have been adopted to XPDL, making the specification very large and hard to overview: The XPDL’s XML Schema Definition is about 110kB in size, compared to 25kB in the case of BPEL.

An evaluation of XPDL with respect to its expressiveness when modeling a number of workflow patterns \[2\] was rather disillusioning. It seemed that the expressiveness of XPDL was “the intersection rather than the union” of that of other workflow notations. Being used as an interchange language the modeler would have to constrain his diagrams to those elements supported by XPDL (and, in case XPDL offers more than a given notation, it would not be possible to transfer an XPDL diagram to this notation).

In its latest version XPDL is intended to be the syntactical counterpart to the graphical notations defined by BPMN. While the idea behind XPDL clearly is ambitious it suffers from a inflated notation and complicated syntax.

### 2.2.5 Simple Notation

In a thought experiment Havey \[15\] states that even a notation like BPMN might not be simple enough to be understood by most business users. While it might be possible for them to understand most of a diagram drawn using BPMN they wouldn’t be able to modify or create one on their own. There are too many details, such as constraints and restrictions on the use of some elements in certain situations, which are not obvious on first sight, so having the business users participating in drawing such a diagram would need too much guidance by the professionals.

Havey proposes to use a Simple Notation for the requirements analysis, which is very intuitive to understand. Machines, on the other hand, won’t be able to understand this notation, thus the models will have to be translated to a more formal notation later on.

The Simple Notation is made up of only four elements: Boxes and arrows for the concepts of activities and control flow. Further there are annotations (which are
2.3. Comparing BPMN and UML Activity Diagrams

represented as plain boxes, too) and associations, e.g. to bind an annotation to a node. There are no restrictions on how to label the nodes and edges, since everything will be intuitively to understand by humans. See figure 2.6 for how the e-mail example from earlier in this section could look in Simple Notation.

Figure 2.6: E-mail example in Simple Notation

These few elements are enough to model the basic workflow patterns such as decisions and loops. However, for modeling more complex situations like reacting on events or more sophisticated control flow rich use of annotations will be necessary, making diagrams drawn in the Simple Notation intuitively but weary to understand. Further some diagrams might be ambiguous and not clear to understand by other readers – let alone machines.

2.2.6 Proprietary Notations

Many modeling tools for the Business Process Execution Language (BPEL) provide their own graphical notation for BPEL. Making use of the fact that most elements of BPEL are block-structured the resulting diagrams often look like a mixture of BPMN and Nassi-Shneiderman diagrams. While these diagrams are of course very close to the executable code, making it easy for BPEL experts to write and maintain processes, it is obvious that this type of diagram is not suited for exchange among different departments and institutions.

2.3 Comparing BPMN and UML Activity Diagrams

From the above, UML Activity Diagrams and BPMN obviously are the best choices. Both are standardized, formal and not too difficult to understand. But while creating models using UML has taken its place in the software development lifecycle BPMN still is a new technique. In this section we will compare these two notations.

White [37] compared BPMN and UML Activity Diagrams by testing their capability of describing 21 workflow patterns[1]. Most of these patterns can be drawn nearly equally with both notations, but while in BPMN often there are several possibilities of drawing a certain pattern Activity Diagrams are more restrictive. He explains the similarity of both notations with the fact that both are designed to describe the same domain. He writes that they even might “converge” some day. The first step, the adoption of BPMN by the OMG, is already done.

UML is a very large specification with many different diagrams, each showing only one view on the model. Each diagram has its own notation and rules. Elements,

\[1\] meanwhile the list has been extended to 43 patterns, see [30]
such as arrows of some line style, have different meanings in different diagrams of UML. In BPMN there is only one type of diagram providing one view on the system, but this one view is more complex, providing more information than UML Activity Diagrams. With BPMN being used instead of UML Activity Diagrams the process models will be more precise. BPMN will never replace UML, but it is a good addition.

While UML is the more complete modeling notation BPMN has the advantage that all the information is caught in a single diagram. BPMN is not suited for modeling data structures and data flow, but for processes it’s more powerful than UML Activity Diagrams.

With its subtypes and attributes a diagram written in BPMN is providing more information than a similar UML Activity Diagram. This information can be used for a mapping (and thus for exporting) to a executable language. Depending on how accurate the model can be mapped to the target language, the mapping could be rough, regarding the basic types only, or very accurate, finding an equivalent for each attribute and diagram pattern, thus BPMN diagrams could be mapped to instantly executable programs.

The best choice for the requirements analysis might be to use a simplified version of the BPMN, with only the basic elements and some simple to understand subtypes such as timer events or standard loops, or to change to a more simplified notation in case the business users show too few understanding. In both cases the process model will have to be refined or even redrawn by the professionals after the requirements analysis. From this point on it can be used throughout the further design. After the design phase it can be transformed to an executable language, being the fundament for the final implementation.

2.4 A Closer Look at BPMN

After the short introduction earlier in this chapter we will use this section to take a closer look at BPMN, since this will be the process modeling language used in this thesis. More details and advanced modeling techniques can be found in the full specification [27].

2.4.1 BPMN Elements

This section is intended to give a brief introduction of each of the basic element groups:

- **Flow Objects**: Events (Start, End and Intermediate), Activities (Task and Subprocess) and Gateways

- **Connecting Objects**: Sequence Flows, Message Flows and Associations

- **Swimlanes**: Pools and Lanes

- **Artifacts**: Data Objects, Groups and Annotations
Flow Objects

The category of Flow Objects, the most important elements in BPMN, is made up of Events, Activities and Gateways. All Flow Objects are held in Lanes (see below).

Events are things that happen, like a message arriving, an alarm, or an error, and often they mark the beginning and the end of the process. The graphical notation is a circle. They are subdivided into Start Events, Intermediate Events and End Events, which will determine the circle’s border (see figure 2.7).

Figure 2.7: BPMN Event types. From left to right: Start Event, Intermediate Event, End Event

Further all three Event types have a variety of subtypes which will determine e.g. a Start Event’s trigger and an end Event’s result. Each of these subtypes can be distinguished by a different icon in the center of the Event figure (see figure 2.8) and results in a number of attributes to be set for the Event.

Figure 2.8: BPMN Event sub types. From left to right: Rule, Timer, Message, Link, Multiple, Cancel, Compensate, Error, Terminate

Basically, an Activity is something that is done. Activities subdivide in Tasks, which are atomic Activities, and Subprocesses, which are composite Activities. The graphical notation for an Activity is a rounded rectangle with the Activity’s name inside of it. Subprocesses are marked with a small ⊞ sign on the bottom line (see figure 2.9).

Figure 2.9: BPMN Activity types. From left to right: Task, Subprocess

Like Events, Activities also have some specializations, each one with special attributes: They can be for instance a Send or a Receive Task, stand for some Manual work to be done, execute a Script or, in case of the Independent Subprocess, represent a whole business process, just to name a few. All of these subtypes have the same graphical representation, but modelers and modeling tools are free to extend the diagrams with additional markers for the subtypes.

What makes Activities stand out from the other Flow Objects is that they can loop. Although in BPMN loops also can be defined by simply connecting a Sequence Flow to an upstream Flow Object, which might be easier to understand by non-experts,
it’s seen as better style to use looping Subprocesses. A looping Activity is marked with a small counter-clockwise arrow on its bottom line.

**Gateways** provide wide capabilities in modeling all kinds of splitting and merging behavior. Figure 2.10 shown the different kinds of Gateways. Depending on whether the Gateway has multiple incoming or outgoing Sequence Flows – or even both – it has different semantics, like forking and/or joining the flows. However, Gateways are not the only way for modeling forking and joining of flows. In some cases the same semantics can be reached by omitting the Gateway and connecting multiple Sequence Flows directly to an Activity.²

Figure 2.10: BPMN Gateway types. From left to right: Data based XOR (with and without marker), Event based XOR, Inclusive OR, Complex, AND

**Connecting Objects**

The most important connections are **Sequence Flows** and **Message Flows**. Sequence Flows represent the flow of control and connect Flow Objects within a Pool in the order of execution. Message Flow represents messages – not necessarily data – being exchanged exclusively between Pools. See figure 2.11 for the connections’ graphical notation.

Figure 2.11: BPMN Connection types. From left to right: Sequence Flow, Message Flow, Association

The third connection, the **Association**, is mainly used for documentation reason, for instance to connect a Text Annotation to a Flow Object that needs further explanation. Still there is an exceptions to this rule: For connecting a compensating Activity to a compensation Event an Association is used instead of a Sequence Flow. An Association’s arrow heads are optional.

**Swimlanes**

Swimlanes can be **Pools** and **Lanes**. Each Pool represents one Participant in the business process, while Lanes are used to partition a Pool. Doing so each of a company’s departments could be represented by a Lane while the Pool stands for the company itself. Typically Pools and Lanes are oriented horizontally, but they may be oriented vertically, too. Further, Lanes may cross or contain other Lanes, but those techniques are poorly documented and usually not used. Figure 2.12 shows a small horizontally oriented Pool containing two empty Lanes.

²this is not allowed for Events
2.4. A Closer Look at BPMN

Figure 2.12: BPMN Swimlanes. A Pool with two Lanes

Artifacts

The main purpose of Artifacts is documentation. However, like Associations, in some situations they can have semantics, e.g. when a Data Object is referenced by an Activity as input. Data Objects represent everything that can be input or output of some Activity. In most cases this will be a file, but since Activities can be Manual Tasks, too, a Data Object could also stand for something physical.

The other two Artifacts, Group and Text Annotation, are solely used for documentation. See figure 2.13 for their graphical notation.

Figure 2.13: BPMN Artifacts. From left to right: Data Object, Group, Text Annotation

The specification states that this category may be extended by proprietary artifacts which could have more semantics, too. This way BPMN can be extended with new elements to represent concepts that were not considered in the original specification.

2.4.2 Levels of Complexity

BPMN can be seen as having at least three levels of complexity.

1. Basic Types: All diagrams are made up of the basic elements of the four categories: Events, Activities, Gateways, Connections, Pools and Artifacts. These can be understood easily even by non-experts.

2. Subtypes: The Flow Objects each have several subtypes, e.g. Timer Events, Receive Tasks and Inclusive Gateways. Using the same shapes as the basic elements enriched with some additional graphical information, like an icon, the symbol’s basic type can be clearly identified by non-experts while providing additional visual information for the professionals.

3. Attributes: Each of the BPMN element types provides a large number of both primitive and complex attributes. While some of these attributes are visible in the diagram, like a flow object’s name or subtype, most are non-graphical. These attributes enrich the diagram with the formal semantics necessary for the export to an executable language while not polluting the visual notation with too many details.
If the process shall be mapped to a executable language the third level is very important: Not only does it give values for many attributes that otherwise would have to be set manually. Some of the visual elements of the BPMN do not have semantics “on their own”. Message Flows for example do not have a mapping to WS-BPEL. Instead the source Activity has to be of type \texttt{Send} and the target Activity of type \texttt{Receive}, and both have to reference the same non-graphical \texttt{Message} element. This is necessary in cases when the communications partner is not in the same diagram and thus a Message Flow can not be drawn.

Of course it is free to the designers of new mappings to map the Message Flow, if it is available, without insisting on the existence of the non-graphical Message element.

### 2.4.3 Export and Code Generation

One of the main purposes of the Business Process Modeling Notation is to provide a graphical notation that can be used to generate executable code from it.

A mapping to WS-BPEL is given in the BPMN Specification \cite{BPMN27, Chapter 11}. As a matter of fact the BPMN has been tailored for the mapping to WS-BPEL, which can be seen in many attributes which are needed only for the mapping. Most of these attributes can be reused for mappings to other languages, too, e.g. such common concepts as properties and assignments.

On the other side BPMN has more expressive power than BPEL. A diagram in BPMN is a directed graph, while BPEL (and in fact most other executive languages as well) is block oriented, making the export to a semantically equivalent program somewhat complicated and in some cases impossible. Numerous papers have been written on how to identify block structures within a BPMN diagram or how to alter an existing diagram to conform to block structure. However, not every diagram can be refactored like that.

While the basic elements such as Flow Objects shall neither be altered nor extended by new elements the BPMN Specification encourages the introduction of new, domain-specific Artifacts to be used in mappings to executable languages other than BPEL. These elements can be associated with the original BPMN elements and represent concepts that were not considered in the original BPMN specification.

In this chapter a number of process modeling notations have been introduced and the core elements and features of the Business Process Modeling Notation have been lined out. The next chapter will briefly introduce the basic concepts of Model Driven Engineering and how they can be applied for generating powerful domain specific editors. Further an overview on known problems and existing approaches concerning the model driven generation of executable programs based on process models will be given.
3. Model Driven Engineering

Model Driven Engineering (MDE) and OMG’s Model Driven Architecture (MDA) \cite{23} are intended to improve and accelerate software development. The motivation is that software development should focus on a program’s specification instead of the implementation. By extensive use of generators the code is correct, quickly produced, reusable, conforming to standards and of high quality.

In Model Driven Engineering a number of models are used for different views on the system, each with a different level of abstraction.

1. The Code can be seen as the lowest level of abstraction.

2. A Platform Specific Model (PSM) is abstracting from the program code while preserving its basic concepts, like classes and methods. A popular example for this kind of model is UML Class Diagrams.

3. The Platform Independent Model (PIM) is giving a very abstract view on the system. Examples for platform independent models are for instance some UML diagrams, like UML Usecases and Activity Diagrams and BPMN.

In Model Driven Engineering the platform independent model (PIM) is used for modeling the program’s functionality from a high level view (“programming in the large”). This model is especially useful to involve stakeholders and domain experts in the development. The PIM then can be used to generate platform specific models (PSM) for further refinement and completion, while non-functional, implementation specific details, such as exception handling and resource management, do not have to be taken into account. Finally the program code is generated from the PSM. In the ideal case no manipulation of the code is necessary; it can be fully generated from the higher level models. Some tools also provide round tripping, meaning that changes made to the code are automatically transferred to the models, keeping them up to date.

When the project has to be modified, in MDE it is best practice to change the models and to regenerate the code. This way MDE encourages incremental and iterative
design and the model is kept up to date, as shown in figure 3.1. Further the model is independent of platform and programming language. The same model can be used to generate an equivalent program in a different language, making programs developed using MDE highly portable: When a company has to change to a different programming language or framework the new software can be generated from the same models.

MDE might look like an unnecessary additional effort. Creating the different models is time-consuming and understanding the generated code and the interaction of the several generated modules can be much more difficult than understanding code written by oneself. But in projects with many developers MDE can help to provide a consistent coding style and program structure that can be understood by everyone involved in the project.

The editor developed in this thesis is about Model Driven Engineering in two ways:

1. The editor has been designed and implemented using MDE.
2. The editor can be used for developing programs using MDE.

The following sections will describe how Model Driven Engineering can be used for the quick creation of domain specific editors using the Eclipse framework. Further existing approaches on the generation of executable programs with BPMN diagrams will be introduced as well as known problems that arise when transforming unstructured workflows to block structured programs.

### 3.1 Using MDE for Designing Eclipse Editors

Not only can Model Driven Engineering be used for creating programs that were fully specified in a model. Another popular field of application for MDE is the generation of domain specific editors. In this case the modeling tool can be specialized on the creation of editors and thus provide a variety of modules and tools that can easily be combined to the model for a feature-rich editor for a given domain. In the Eclipse Modeling project (EMP) there are several plugins supporting the model driven development of other Eclipse editor plugins by providing both tooling and runtime support for them.

Of course one advantage of model driven architecture, the independence of a programming language, does not apply in this case, since all the programs developed
3.1. Using MDE for Designing Eclipse Editors

This way will always be Java programs and depending on the Eclipse framework. However, Java runtime environments and the Eclipse IDE are available for every platform. And of course the result basically has to be an editor for some domain. But the domain and most of the editor’s functionality can be specified with a set of models.

The frameworks introduced in this section also can create a Rich Client Platform (RCP) Application which can be used without having Eclipse installed and running. For this purpose Eclipse can spawn a minimum copy of itself, holding only those plugins that are required for launching the editor. Such a RCP application can have a more individual look than a plugin and the platform is much smaller in size. However, this way the editor will not be capable of editing anything else than this editor’s domain. The best practice might be to deploy the editor both as a plugin and a RCP, so those already having an Eclipse platform in use can use the plugin while others can use the RCP application.

3.1.1 The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [7] is an implementation of Essential MOF (EMOF). An EMF model (“Ecore”) can be generated from a UML class diagram, annotated Java interfaces or XML and in return can generate all these. From this model not only the implementations for the specified model elements can be generated, but also a full editor (see figure 3.2). A good introduction to EMF can be found in [5].

Figure 3.2: Creating an EMF model from other models and generating code

EMF Generated Models

The generated implementations provide many features.

- automated serialization
- implementation of the Observer pattern
- undoable editing
- multiple inheritance
- merging of generated and handwritten code
- generic access to classes and attributes
- dynamic creation of new classes at runtime
By default the model instances can be saved persistently to XML. If the Ecore has been generated from an XSD file the serialized form will have exactly the syntax specified in the XSD. However it is also possible to provide another serialization that can be integrated in the framework as well.

Extending the class `EObject` the model instances will notify their `Adapters` whenever they undergo changes, e.g. when an attribute is assigned a new value. This not only enables refreshing diagrams as needed; additionally every change to an EObject can be recorded, providing undoable editing.

Since for each class in the model there is an interface and an implementation inheritance EMF models provide multiple inheritance: Several interfaces are implemented and the generator is taking care for the code duplication necessary since Java itself does not support multiple inheritance.

The generated code can be altered and extended with handwritten fields and methods. Using annotations the generator recognizes which parts shall be regenerated and what has to remain untouched.

Apart from the actual interfaces and implementation for all the classes defined in the model there will also be a single `Package` class with a constant internal instance for each of the classes, attributes and references in the model (see figure 3.3).

![Diagram](image.png)

Figure 3.3: The Ecore model’s internal representation (simplified subset)

Using these “meta classes” EMF is providing reflective access to all the classes and attributes (introspection). Thus instead of `somePerson.setName("Peter")` the reflective version `somePerson.eSet(FooPackage.PERSON__NAME, "Peter")` can be used. While not needed in general this can be useful for dynamically creating new classes at runtime:

```java
EAttribute name = EcoreFactory.eINSTANCE.createEAttribute();
name.setName("name");
name.setEType(EcorePackage.eINSTANCE.getEString());

EClass person = EcoreFactory.eINSTANCE.createEClass();
person.setName("Person");
person.getEStructuralFeatures().add(name);

EPackage myPackage = EcoreFactory.eINSTANCE.createEPackage();
myPackage.getEClassifiers().add(person);

EFactory myFactory = myPackage.getEFactoryInstance();
EObject somePerson = myFactory.create(person);
somePerson.eSet(name, "Peter");
```
EMF Generated Editors

Apart from a feature rich and efficient implementation of the domain model the EMF framework also can be used for generating a complete editor for it. This can be done with literally one click. The generated editor provides a semi-visual tree view for the model, undoable editing, serialization and deserialization. Like the model files the editor can be customized if necessary, too.

As said before an RCP application can be created, too. With the EMF generator and EMF plugin development assistants this is a matter of minutes. The resulting RCP application’s size is only about 10% of the original Eclipse application’s size.

3.1.2 The Graphical Modeling Framework

The Graphical Modeling Framework (GMF) [11] is a promising approach to provide a framework for generating standardized feature-rich visual editors for EMF models.

A brief introduction to the GMF runtime can be found in [29, 33]. The first version, GMF 1.0, was released with the Callisto Simultaneous Release in June 2006 and version 2.0 will be part of the Europa Simultaneous Release scheduled for June 2007.

The Graphical Modeling Framework consists of two parts: The GMF tooling and the GMF runtime. The tooling part is providing a modeling framework for designing new editors and the runtime part eases the integration of EMF and GEF, the Graphical Editing Framework [10] used for the editor. Additionally it receives contributions from the Eclipse Modeling Framework Technology (EMFT) project and uses EMF OCL, EMF Validation, EMF Query and EMF Transaction.

Tooling

The tooling part is providing a set of wizards and editors for modeling the editor (see figure 3.4) that will then be generated and use the runtime part.

![Figure 3.4: The GMF Dashboard showing the various models of the GMF tooling process](image)

The models and editors used for designing the GMF editor are in fact created with EMF. These are the models: Graphical Definition Model, Tooling Definition Model, Mapping Model and Generator Model. One more model, the Notation Model, is used in the GMF runtime part.
For modeling a graphical GMF editor one starts with the editor’s domain as an EMF model. Next the visual representation is defined: A figure gallery with ovals, rectangles, lines and decorators, which then are associated to nodes and connections. The third part in the GMF tooling is defining the tools to be used. Basically this will be a palette with entries for each node and connection, but also additional menus can be defined. In the last part the domain model, the notational model and the tooling model have to be glued together in the mapping model. Additional constraints for validation can be defined here, too. Finally the generator model is created from the mapping model. It is possible to define details like file extensions, names, layouts or whether to create a plugin or a RCP application. Normally this does not require any modification by the user and the default settings can be used as well. Now the visual editor can be generated and run.

By storing the various aspects of the editor – the graphical definition, the tooling definition and the mapping – in separate models each part can be reused without the others. For instance it would be possible to provide different GMF editors for different views on a single domain. Each editor would use the same underlying Ecore model, graphical definition model and tooling model, but different mapping models. Further the figures designed in the graphical definition model can be exported to a separate figures plugin and reused with any GEF based editor.

**Runtime features**

Aside from facilitating the integration of EMF and GEF the GMF runtime is providing many value-rich features that otherwise would have to be implemented every time anew, such as:

- a node’s compartments can be collapsible, so the user can hide details
- direct editing support for all labels in the diagram
- modeling assistants: popup-bars and connection handles which provide fast access to all the model elements that can be used in the current context
- common tools, menus, toolbars and properties, like sticky notes, zooming, coloring, arranging, auto size, printing, clipboard support, graphics export and many more

Aside from the easy generation with the GMF tooling and the many features that are provided by the GMF runtime there are also more general benefits in using GMF.

An application created with GMF automatically has a look and feel familiar to every user of the Eclipse framework. And not only the user but also the developer will profit from the unified structure of the generated editor. When first generating a GMF editor one might be discouraged by the somewhat bulky code. This is just natural and not different from when someone starts maintaining code originally written by someone else. However, since the code of each editor generated with GMF has the same structure, once familiar with GMF maintaining a GMF project will be easy, even if it was originally designed by someone else.

In the default configuration the editor will save diagrams in two files: One file for the model, the other for the layout information. This is where the last model...
mentioned earlier, the notation model, is applied. This model is holding all the layout information, like positions, colors and fonts. This way the model does not have to be enriched with layout information, like a point for a node’s location or a list of points for a connection’s bendpoints. This way the pure model file can be interpreted by other programs, too, especially when generated from a XSD. Further a new layout file can be initialized from the plain model file.

On the downside each model element has to be synchronized with it’s View element when being created, moved or deleted. In GEF the only thing needed to insert a new child node is to create the node’s model, set it’s coordinates, insert it into its parent and make a call to refreshChildren. In GMF the same task is more complicated.

Last not least an editor created with GMF will be maintained and enhanced even when actually no one is maintaining it: The GMF community is continuously improving the GMF framework, adding new features and fixing bugs. When a new version of GMF is available the editor can be regenerated with that new version and it will provide all the new version’s features and bugfixes.

Like in EMF the generated code can be customized in many ways. When regenerating the editor due to changes in the model the generated code will be merged with those parts written by hand.

### 3.2 Using MDE for Modeling Processes

In this section we will look on the second aspect of Model Driven Engineering covered in this thesis: The model based creation of arbitrary programs, especially services and business processes. First we will present the major challenges in transforming BPMN diagrams to executable process models. Thereafter we will present some existing approaches and tools.

#### 3.2.1 Unstructured Workflows

The major challenges in transforming a BPMN business process diagram to a program arise from the fact that, while a BPMN diagram is a unstructured directed graph, most executable languages are based on structured workflows. Thus the source model has to be altered so it can be expressed in a structured form. This is not always possible, since without the restriction to structured models the source language, BPMN, has more expressive power than the target languages.

A good definition of *Structured Workflow Models* (SWMs) is given in both [17] and [22]. The definition says the following:

1. A single activity \( A \) is a SWM. \( A \) will be both the initial and the final activity of the SWM.

2. Two SWMs \( X_1 \) and \( X_2 \) with a control flow going from the final activity of \( X_1 \) to the initial activity of \( X_2 \) are a SWM (a sequence).

\[^1\text{while this might still be applicable for simple layouts where each node has a position and each connection some bend points, it is not for providing e.g. persistent colors and fonts}\]
3. A block starting with an and-split (or-split) $S$ and ending with an and-join (or-join) $J$ with a number of SWMs $X_1$ to $X_n$ in between, with control flow going from $S$ to the initial activities of each $X_i$ and from the final activity of each $X_i$ to $J$, is a SWM (a parallel or alternative block).

4. A block starting with a or-join $J$ and ending with a or-split $S$ and two SWMs $X_1$ and $X_2$ with control flow from $J$ to the initial activity of $X_1$ and from the final activity of $X_1$ to $S$ and from $S$ to the initial activity of $X_2$ and from the final activity of $X_2$ to $J$ is a SWM (a loop).

Please have a look at figure 3.5 for a more descriptive picture of Structured Workflow Models.

Figure 3.5: Basic Structured Workflow Models: Atomic Activity, Sequence, Block, Loop. Darkened nodes stand for composite parts of the workflow that have already been identified as a SWM.

The fact that a structured workflow model has less expressive power than an arbitrary workflow and that not every arbitrary workflow can be mapped to a semantically equivalent structured workflow can be made clear by the following: The definition of structured workflows guarantees that structured workflows do neither deadlock nor result in multiple instances of the same activity being active at the same time (also referred to as a lack of synchronization). Arbitrary workflows like BPMN diagrams can deadlock or result in multiple instances. Thus a structured workflow does not have the same expressive power as an arbitrary workflow.

The two kinds of structural conflicts mentioned above – deadlocks and a lack of synchronization, resulting in multiple instances – can occur when merging sequences with an and-join, which can result in deadlocks if not all these sequences are visited in the workflow instance, or when merging sequences with a or-join, which can be the cause for a lack of synchronization, if more than one of the sequences are visited in the workflow instance.

For the transformation of a unstructured workflow (1) structured workflows have to be recognized in the directed graph and (2) if the directed graph is not a structured workflow it has to be altered accordingly.

Validation and Identification of Structured Workflows

One possibility for validating whether a directed graph is a structured workflow is to create a set of rules based on the definition given earlier in this section. Firstly every atomic activity is associated by a reference object. Then rules for identifying sequences, blocks and loops are applied as long as possible in a bottom-up way, reducing the involved activities’ references (or the graph itself). If in the end only
one workflow reference is left – that is, if all the smaller structured workflows are successfully combined to one – the diagram as a whole is a structured workflow.

In [31, 32] a set of rules is introduced for identifying not only workflows that follow a block structure but also workflows that are acyclic directed graphs while not having deadlocks or a lack of synchronization. The rules are applied iteratively, reducing the workflow by simplifying those parts that are definitely correct, for instance when there is a node with exactly one incoming and outgoing control flow the node is removed and the flow spans from the node predecessor to it’s successor.

If the workflow can be reduced to a trivial workflow (that is, an empty or atomic workflow) then the workflow is correct in the sense that it is free of structural conflicts. Otherwise, if the workflow can not be reduced to a trivial workflow, it must contain at least one structural conflict. The algorithm has been implemented in a tool that can import diagrams from IBM’s MQ Workflow; validate and re-export them.

In [21] it has been shown that the algorithm is not complete and that there are correct workflow models that can not be completely reduced by the algorithm. The algorithm has been extended and its correctness and completeness have been proved.

**Transformation of Unstructured Workflows**

The transformation of unstructured workflows to structured workflows is much more complicated than identifying existing structured workflows (and transforming them to other workflow languages).

The reason is that there are a lot more patterns of unstructured workflows than for structured workflows. As written before all structured workflows following certain criteria can be identified with some simple rules. In [17] and [22] various patterns of unstructured control flow have been identified: Entry in and exit from a block (loop) structures, overlapping blocks (loops), one split (join) for multiple joins (splits), and many others. Approaches for structuring these patterns were given, too. However these solutions are quite complicated and it is not known to the author in how far these approaches have been implemented yet.

Further it has been shown that while each unstructured workflow without parallelism can be translated to a structured workflow the introduction of parallelism can cause both deadlocks and multiple instances and that there are arbitrary workflows that can not be modeled in a structured way at all [17, Theorem 2]. In some languages, such as BPEL, multiple instances can be achieved by spawning processes, but it is obvious that such programs will be hard to understand and to maintain. Further BPEL’s link element can be used for modeling some sorts of overlapping parallel structures.

Despite these exceptions every arbitrary workflow model has to be changed to a structured workflow model before being mapped to a structured target language. Basically this can be done by the modeler as well, but it is much more convenient to have this step automated.

As stated in [17] the transformation of a unstructured workflow to a structured workflow can require node duplication and the use of auxiliary variables. In the case of node duplication an activity or a whole workflow segment has to be duplicated
and placed in two concurring branches of the structured workflow that were interconnected in the unstructured workflow. When using auxiliary variables a new variable, usually of type `boolean`, has to be declared. The variable is set in one branch of a concurrency structure and can later be read, remembering in which branch the workflow was earlier (and, in the unstructured version of the workflow, still would be in).

Both approaches make the workflow very hard to understand and to maintain. Thus it is advantageous not to restrict the user to create structured workflows but to allow both structured and unstructured workflows and to transform unstructured workflows to an equivalent structured workflow model later.

For the validation of the resulting workflow a *bisimulation* can be made, meaning that both the source and target workflows are simulated manually or programmatically. If in each situation the target workflow can simulate the source workflow\(^2\) and vice versa then the workflows are equivalent.

In [18] another rule-based approach for handling unstructured cyclic flows, that is, cyclic workflow structures with more than one entry and / or exit points, is presented. The approach is based on a method that has been introduced about forty years ago and was originally invented for the elimination of `goto` statements in compiler theory.

By understanding control flow in workflow diagrams as just another form of `goto` statements the idea could be transferred to cyclic workflows. After transforming an unstructured cyclic example workflow to a structured cyclic workflow a possible application on a BPEL workflow containing invalid cyclic links has been shown. The authors state that due to the simplicity of their algorithm the restriction to acyclic links in BPEL could be dropped entirely.

### 3.2.2 From BPMN to BPEL

The motivation of BPMN was to facilitate the creation of business processes and to put an end to the numerous proprietary process modeling notations used in various BPEL process design tools. Thus the BPMN specification already comes with a recommendation on a mapping of BPMN to BPEL [27, Chapter 11]. Still this mapping does not cover all aspects of BPMN and there are elements and attributes that can not be mapped to BPEL, such as Transactions and Ad-Hoc Processes.

In the mapping proposed in the BPMN specification control flow modeled with Sequence Flows is mapped to structured elements, such as BPEL Sequences, Flows and While-Loops. Note that there are other proposals for mappings, too. White, the author of the original mapping, presents an alternative mapping [38]. In this mapping all elements are contained in a single Flow element. Sequence Flows are mapped to Links, connecting the Activities. Only the While element is still needed for loops, since Links must not be cyclic in BPEL.

In [28] another extension of the original mapping from BPMN to BPEL is shown. The authors show that some otherwise problematic workflow patterns, such as arbitrary cycles and multiple instances, can be mapped to BPEL using the rarely used

\(^2\)meaning that when the source workflow is in state A and has active transitions to the states B, C and D then the target workflow also has to be in state A with active transitions to exactly the states B, C and D
3.2. Using MDE for Modeling Processes

Event Handler element. Extensions to other patterns such as deferred choice and cancellation are planned.

A list of existing implementations of BPMN can be found at [http://www.bpmn.org/BPMN_Supporters.htm](http://www.bpmn.org/BPMN_Supporters.htm). Note however that not necessarily all of those implementations also provide export functionality.

### 3.2.3 From BPMN to Agents

The transformation of BPMN or other process modeling notations to agent concepts is a relatively new field of research. To the author’s best knowledge only few approaches have been made so far.

In [12] we made the first step towards a transformation from BPMN to the JIAC framework, by giving a formal definition of a *BPMN Normal Form* and a mapping from this normal form to petri nets, which then can be validated with respect to termination, relevance of places and boundedness.

Following a different approach, Vidal et al. [34] propose to replace the static BPEL workflows used today by those dynamically composed by an AI planner and executed by agents. For this each web-service would need to have a machine-readable description of its precondition, effect, input and output, for instance based on the Web Ontology Language for Web Services (OWL-S) [6]. As an example the planning of a vacation is given, including the booking of a flight and hotel, using existing web services but without requiring the user to care about all the details. Still the computational complexity is a problem, especially in the case of conditions and loops, when the plan would have to be reconsidered. Another possibility would be to describe the rough workflow using BPEL or the likes, using agents and planning to care only about the details, like which weather service to choose, e.g. based on other agents’ ratings.

In this chapter the basic concepts of Model Driven Engineering have been introduced as well as how they can be applied for the creation of editors and the generation of executable programs from workflows. In the next chapter some transformation tools which can be used for that purpose will be introduced briefly. The EMT framework, being used in this thesis, will be described in detail.
4. Model Transformation Tools

In this chapter we will have a look on existing tools providing the kind of model transformation needed for the mapping from BPMN to BPEL and JIAC. As we will see there are a number of frameworks for model transformation based on the Eclipse Modeling Framework (EMF). We will introduce these frameworks briefly. Then we will have a closer look at the tool used for the mappings in this diploma thesis.

4.1 Eclipse Generative Modeling Tools

In the Eclipse Generative Modeling Tools Project (GMT) [9] there are a number of sub-projects about Model Driven Engineering and model-to-model and model-to-code transformation. Note that there are other transformation frameworks for other meta models than EMF, but since EMF is very well integrated in Eclipse and most of the workflow models used in this thesis are designed in EMF we will concentrate on this framework.

ATL [16] is a model transformation system developed as part of the ATLAS Model Management Architecture. It is well integrated into the Eclipse platform, providing its own editor, compiler, debugger and virtual machine. In ATL a transformation can be defined by a set of textual rules.

There are some other transformation frameworks, such as MTF\(^1\) and Tefkat\(^2\) similar to ATL. They all work on EMF and use textual rules for defining the mapping from the source model to the target model.

However, the examples given in the transformation systems’ documentations are limited to straightforward transformations, like from Classes to Tables, from Students to Persons or from arabic to roman numbers. It is not clear in how far these transformation systems are suited for the transformation of workflow models.

Another promising approach is that of Open Architecture Ware (oAW), another subproject of the GMT project. Similar to the above, rules can be defined in its own

\(^1\)http://www.alphaworks.ibm.com/tech/mtf
\(^2\)http://tefkat.sourceforge.net/
textual syntax, but it can also integrate rules from other transformation systems and operate on different metametamodels. It can be used for validation, endogenous and exogenous model transformations and code generation, including the generation of a full-featured source code editor for a given BNF.

4.2 The Tiger EMF Transformation Project

The Tiger EMF Transformation Project [14], also referred to as EMT, is a model transformation framework for EMF models developed by the TFS Group at TU Berlin. Similar to the Attributed Graph Grammar System (AGG) [13] the transformation is defined by graph transformation rules.

The main purpose of EMT is to provide a framework for defining endogenous rules, e.g. for the refactoring of models of only one language, which is intended to be used for complex modeling operations and model refactorings and optimization, as can be seen in [4]. However, in its latest version EMT is also suited for exogenous transformations, i.e. model-to-model transformations.

In contrast to many other model transformation tools EMT comes with a visual editor as shown in figure 4.1. Using this editor facilitates the creation of rules greatly. The editor has recently been implemented in the course of a diploma thesis [19] at the TFS Group.

![Figure 4.1: The EMT Visual Rule Editor showing a rule of a transformation model for mapping activity diagrams to petri nets](image)

The following sections will introduce you to the most important concepts of the EMT transformation framework.

4.2.1 Transformation Rules

The transformation rules consist of a left hand and a right hand rule side (referred to as LHS and RHS) which are both models of some language. The LHS serves as a pattern that has to be found in the source graph to be transformed. Elements of the LHS can be mapped to elements of the RHS of the same type. Those elements then can be modified by the rule. Elements of the LHS which do not have a mapping to
the RHS are deleted and vice versa elements in the RHS that are not mapped to by the LHS are created in the course of the rule’s execution.

Besides these most basic concepts the rules can have negative application conditions (NACs) and variables. Variables are needed to refer attributes of elements from the LHS in the RHS and to define attribute conditions in the LHS. NACs are patterns that must not apply for the rule to be executed. Similar to the RHS, elements of a NAC can be identified with elements of the LHS. If the pattern given in the LHS is found but also the pattern of one of the NACs is found with the mapped elements being the same as in the LHS then the rule may not be applied.

The rules are executed in random order by default. To assure that a rule is not executed before another rule has been applied as often as possible the rules can be subdivided into layers. The rules of each layer will be executed as long as possible before changing to the next layer.

4.2.2 Backtracking and Queries

The pattern matching needed for the application of the rules is done by a backtracking algorithm. Each symbol in the rule’s LHS is represented by a Variable\(^3\) which can hold a number of Queries. There are different kinds of queries for different constraints that have to be checked: The most important are Source- and TargetQueries, which represent references among elements of a rule side. Further there are InjectivityQueries, which check if two variables are (not) instantiated with the same object, TypeQueries and VariableQueries.

In the backtracking algorithm the first variable is instantiated with a possible value from its domain, the set of instances of the given type contained in the source model. Next the remaining variables’ domains are reduced by evaluating the last variables queries. Only those instances are retained that are not in conflict with any of the queries. If there are possible instances left for the next variable the process is continued until each variable is instantiated and the rule can be applied. Otherwise the previous variable is instantiated with the next value. If no match is found the rule can not be applied.

4.2.3 AGG and EMT

Basically EMT is providing the same functionality as AGG. Still AGG is providing some additional features for the analysis and evaluation of rules. Rules of a transformation can be tested with critical pair analysis and executed step by step. But while AGG is using a proprietary format for type graphs (metamodels) and graphs (instances of type graphs, models), EMT is based on the popular EMF framework, making it more usable in practice.

However, using the EMT framework also brings some restrictions. In an EMT model each element has to be contained in a container, so it is transitively contained in the root element which again is contained in some resource. Having some element in this containment structure referencing another element which is not in a container, and thus not in a resource, the model can not be made persistent anymore. Thus EMT can handle only a subset of the rules possible with AGG.

\(^3\)not to be confused with the “variables” for attribute values
4.2.4 Interpretation and Compilation of Rules

Once the rules have been created in the visual editor they can be either interpreted by the AGG engine or compiled to executable Java code that can be used with no requirements other than EMF.

For the interpretation the EMT rule model is first exported to the AGG metamodel in a straightforward way. The resulting model then can be transformed using AGG and imported back to EMF. This has the advantage that the advanced features of AGG can be used for the evaluation of the rules. Still this approach is not suited very well for practical use.

The rule model can be used for compiling the rules, too, i.e. generating rule classes. This is done using Java Emitter Templates (JET), the generator used by EMF [7]. The compiler creates a pair of Rule and Wrapper classes for each rule as well as a Transformation class providing access to the rules. The wrapper class is providing the information needed for the pattern matching (the LHS and NAC) while the rule class is needed for execution of the rule (the RHS) as well as for undo and redo. Additionally a set of helper classes is generated. These classes are needed e.g. for the matchfinding. They are the same for each set of rules, but generating them together with the rules has the advantage of making the generated classes independent of the EMT plugin.

Both the interpreter and the compiler make use of the EMF metamodel and the reflective API. The transformation model to be interpreted and compiled is a EMF model itself and the code generation is only possible by using reflective methods.

We decided to use the EMT framework for the model transformations to be implemented in this thesis. While being very powerful the framework consists of only about a dozen of classes and has no more plugin dependencies than EMF.

Since the rules are compiled to Java classes they can be modified and extended afterwards. Thus there are no syntactical restrictions to the right rule sides, which can consist of conditions, loops and arbitrary method calls. This is especially important due to the many subtypes of the BPMN elements.

The graph transformation based approach has a clear formal background. With the concept of LHS, NACs and conditions and using EMF’s reflective API even very complex rules involving several interconnected elements can be realized.

In the next chapter the mappings from BPMN to BPEL and JIAC will be introduced. We will explain how the elements of BPMN are mapped to JADL elements and how agent concepts could be modeled using BPMN.
5. The Mappings to BPEL and JIAC IV

In this chapter we will introduce the mappings from BPMN to BPEL and JIAC. While the mapping from BPMN to BPEL is given in the BPMN specification there was no mapping from BPMN to JIAC yet. We will explain how agents could be modeled in BPMN and how the various elements of BPMN can be mapped to JADL.

5.1 BPEL

For the mapping to BPEL we kept close to the mapping proposed in the BPMN Specification [27, Chapter 11]. However, the implementation of the Element Mapping is not yet complete. While the basic concepts of the mapping to BPEL are already implemented there are some features that require a more in-depth knowledge of BPEL, for instance parallel multi instance loops or Timer and Rule Events, which require a separate process to trigger the Event. The mapping of these elements has been put back for now and will be implemented later.

5.2 JIAC

The greatest challenge in finding a mapping from BPMN to JIAC was that the BPMN specification does not include agent concepts.

Though on the first sight it might stand to reason to map Pools to agent platforms and Lanes to agents this approach left many questions open, for instance what the various Flow Objects should be mapped to in this case. The only possibility would have been to map the Flow Objects to service and plan element calls, executed in the order given by the Sequence Flows. Following this approach most BPMN elements would have been mapped to the same JIAC elements, regardless if Sequence Flow or Message Flow or what type of Task or Event.

Another approach was to map Pools to agents, Lanes to planelements of that agent and Flow Objects to the planelement’s control flow. Changing a Lane could then be interpreted as calling a planelement, and returning to the initial Lane would be
returning from that plan. However, there are a number of open questions in the case of forking on one Lane and joining on another. Further the same behavior can be achieved using Subprocesses.

Also it would be possible to provide more than one mapping from BPMN to JIAC. For instance, in one mapping a Process could map to a planelement and Flow Objects to the planelement’s control flown, and in another mapping Pools could be mapped to platforms, Lanes to agents and Flow Objects to planelement calls. However, this approach raises the problem that the interpretation of a given BPMN diagram would be dependent on the specific mapping intended by the modeler.

A possible solution to this problem might be the introduction of custom Artifacts to represent the concepts of agents and platforms.

### 5.2.1 Modeling Agents with Custom Artifacts

The Business Process Modeling Notation may be extended by custom Artifacts. While the existing elements may not be altered and no additional Flow Objects should be introduced, modelers and modeling tools are free to add non-standard elements as Artifacts, which can be connected to the other elements via Associations. Examples for Artifacts that are currently part of the BPMN are Data Objects, Groups and Text Annotations.

It might be interesting for the mapping to JIAC to introduce custom Artifacts. These new Artifacts could represent the concepts of ontologies, agents and agent platforms (see figure 5.1), which do not have a representation in the original BPMN.

![Mockup of custom Artifacts representing agents and agent platforms](image)

Figure 5.1: Mockup of custom Artifacts representing agents and agent platforms

Introducing these artifacts would solve the problems with mapping Pools and Processes to agents: If a Pool is mapped to a single agent this would result in a 1 : 1 relation between agents and plan elements, which is not wanted. However, using Artifacts to represent agents and platforms they can be arbitrarily associated to Pools, allowing $n : m$ relations of agents and plan elements.

### 5.2.2 Element Mapping

The mapping to JIAC presented in this thesis is concentrating on the control flow within a plan element. Pools are mapped to plan elements and Flow Objects to various JADL statements within that plan. Message Flow is mapped to the exchange of messages with speechacts. Lanes do not have any semantics in this mapping. This way it was possible to exhaust the additional information and semantics given in the Flow Object’s subtypes and attributes.
The following tables will give an overview of the current stage of the mapping from BPMN to JIAC. Note that not each element of BPMN has been successfully mapped to JIAC yet, and that not all the mappings proposed in these lists are already implemented in the Visual Service Design Tool. Some of which will require further reconcilement.

### Diagram, Pool and Process

<table>
<thead>
<tr>
<th><strong>Business Process Diagram</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pool</strong></td>
</tr>
<tr>
<td><strong>Process</strong></td>
</tr>
<tr>
<td><strong>Lane</strong></td>
</tr>
</tbody>
</table>

### Events

It is not yet clear how to map Intermediate Events attached to an Activity boundary. Thus the mappings presented here are only for the case when the Events are in normal flow.

<table>
<thead>
<tr>
<th><strong>Start Event</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rule</strong></td>
</tr>
<tr>
<td><strong>Timer</strong></td>
</tr>
<tr>
<td><strong>Message</strong></td>
</tr>
<tr>
<td><strong>Link</strong></td>
</tr>
<tr>
<td><strong>Multiple</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Intermediate Event</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rule</strong></td>
</tr>
<tr>
<td><strong>Timer</strong></td>
</tr>
<tr>
<td><strong>Message</strong></td>
</tr>
<tr>
<td><strong>Link</strong></td>
</tr>
</tbody>
</table>
5. The Mappings to BPEL and JIAC IV

<table>
<thead>
<tr>
<th>Multiple</th>
<th>The mapping for a Multi-Trigger Event could be realized as a <code>par</code> holding the mappings for the other triggers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancel</td>
<td></td>
</tr>
<tr>
<td>Compensate</td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td>This will map to a <code>fail</code> statement.</td>
</tr>
</tbody>
</table>

**End Event**

<table>
<thead>
<tr>
<th>Message</th>
<th>The mapping for a Message End Event should be a <code>send</code> speechact, analogue to the Message Start Event.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link</td>
<td></td>
</tr>
<tr>
<td>Multiple</td>
<td>The mapping for a Multi-Trigger Event could be realized as a <code>par</code> holding the mappings for the other triggers.</td>
</tr>
<tr>
<td>Cancel</td>
<td></td>
</tr>
<tr>
<td>Compensate</td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td>This will map to a <code>fail</code> statement.</td>
</tr>
<tr>
<td>Terminate</td>
<td>This will map to a <code>end</code> statement.</td>
</tr>
</tbody>
</table>

**Activities**

<table>
<thead>
<tr>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Loop</td>
</tr>
<tr>
<td>Multi Instance Loop</td>
</tr>
<tr>
<td>With Event Handler</td>
</tr>
</tbody>
</table>

**Task**

<table>
<thead>
<tr>
<th>Manual</th>
<th>There will be no mapping for a Manual Task.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive</td>
<td>This will map to a <code>receive</code> speechact, similar to the Message Start Event.</td>
</tr>
<tr>
<td>Send</td>
<td>This will map to a <code>send</code> speechact, similar to the Message End Event.</td>
</tr>
<tr>
<td>Service</td>
<td>This could either map to a combination of <code>send</code> and <code>receive</code> speechact or to a service call. In this case the message sent would be the service call and the message received the result of the call.</td>
</tr>
<tr>
<td>Script</td>
<td>This could map to arbitrary JADL code, depending on the content of the given script string.</td>
</tr>
<tr>
<td>Reference</td>
<td>This will map to a copy of the mapping of the referenced Task.</td>
</tr>
<tr>
<td>User</td>
<td>A User Task will require interaction with the user, for instance using a GUI. A marker will be set, remembering the designer to implement the required logic.</td>
</tr>
</tbody>
</table>

**Subprocess**
5.2. JIAC

<table>
<thead>
<tr>
<th>Embedded</th>
<th>The content of the Subprocess will make up a new action element while the Subprocess itself will be mapped to a call element, calling this action.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>Like the Embedded Subprocess this will be mapped to a call element, with the action element already being implemented with the original Subprocess’s mapping.</td>
</tr>
<tr>
<td>Independent</td>
<td>An Independent subprocess could be mapped to a service or planelement call.</td>
</tr>
<tr>
<td>Transaction</td>
<td></td>
</tr>
</tbody>
</table>

**Gateways**

Gateways in general define the extent of blocks and loops. Whether a loop or a block will be created is dependent of the Gateway’s context. For a loop only a XOR-Data Gateway will be allowed. The merging Gateway of both blocks and loops is not taken into account yet.

<table>
<thead>
<tr>
<th>Gateway</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR Data (Loop)</td>
<td>Being part of a loop the Gateway will be mapped to a while, similar to the Activity Standard Loop mapping. The sequence going from the first to the second Gateway, that is, the until-part of the loop, will be copied and inserted before the while. The loop body will be made up of the second sequence, going from the second Gateway back to the first, and the copy of the first sequence. The loop condition will be derived from the condition expressions of one of the Sequence Flows going out of the second Gateway.</td>
</tr>
<tr>
<td>XOR Data (Block)</td>
<td>Being part of a block the Gateway will be mapped to a branch element. The sequences starting at the Gateway will make up the branch cases. A cond will be created. The formulas for the condition will be derived from the outgoing Sequence Flows’ condition expressions. The default sequence will be the last case, being triggered if no other does apply. Another mapping would be to create a alt block with a seq for each sequence starting at the Gateway. Each seq, except that for the default sequence, would start with a eval, holding the condition.</td>
</tr>
<tr>
<td>XOR Event</td>
<td>This will map to a branch element. The sequences starting at the Gateway will make up the branch cases. A receive will be created. The messages for the receive will be derived from the Message Events and Receive Tasks following the Gateway. However, this does not work if one of the Events has a Timer trigger.</td>
</tr>
<tr>
<td>AND</td>
<td>This will map to a par block, holding the sequences starting at the Gateway.</td>
</tr>
</tbody>
</table>
OR

This could be mapped to a \texttt{par} block, holding one \texttt{alt} block per sequence. The \texttt{alt} blocks will each contain a \texttt{seq} starting with a \texttt{eval} and a \texttt{cont}, like \( \texttt{(par (alt (seq (eval [1]) [2]) cont) [3])} \) with [1] being the condition expression of the Sequence Flow, [2] being the elements of that sequence starting at the Gateway and [3] being the other sequences. If there is a default Gate on the Gateway auxiliary variables will have to be introduced.

Complex

Complex Gateways do not have a mapping yet.

Other Elements

<table>
<thead>
<tr>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence Flow</strong></td>
</tr>
<tr>
<td><strong>Message Flow</strong></td>
</tr>
<tr>
<td><strong>Association</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Artifact</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Object</strong></td>
</tr>
<tr>
<td><strong>Text Annotation</strong></td>
</tr>
<tr>
<td><strong>Group</strong></td>
</tr>
<tr>
<td><strong>Custom Artifacts</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Supporting Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Property</strong></td>
</tr>
</tbody>
</table>
### Assignments
Assignments are mapped to `bind` and `unbind` elements. If the from-Expression is not null then a `bind` will be created, binding the term specified in the from-Expression to the variable, referenced by the to-Property. If the from-Expression is null, then a `unbind` will be created for the given variable.

### Messages
Messages are mapped to speechacts when used by Message Events or Send or Receive Tasks. How exactly the various attributes will be mapped to the attributes of the speechact is not decided yet.

### Participants
The Participant could be used to group the planelements their Pools have been mapped to to services and to determine which one will be the user and which the provider.

This chapter explained in detail the present state of the mapping from BPMN to JIAC and presented some ideas how agent concepts, which are not considered in the BPMN specification, could be modeled.

The next two chapters will describe in detail the implementation of the **Visual Service Design Tool**. Firstly we will have a look on the editor, which has been realized using Eclipse GMF. Then we will introduce the implementation of the mapping, describe how the EMT has been used for the transformation and explain the most important rules.
6. Implementation: The Editor

In this chapter we will introduce the implementation of the BPMN model editor, which is part of the Visual Service Design Tool. The implementation of the mappings to BPEL and JIAC will be introduced in the next chapter.

The editor has been realized as an Eclipse plugin, using the EMF and GMF frameworks, introduced in section 3.1. Using these technologies the editor has been specified in a number of models. A large part of the editor code could then be generated from these models, making the implementation as correct as possible and easy to maintain by other developers. Still some customizations were needed to provide all the functionality required for a user-friendly BPMN editor and so all the figures would conform to the graphical notations given in the BPMN specification.

In the following sections we will first describe the rough project structure with its several sub-projects and plugins. Then we will introduce the domain model used for the internal representation of the BPMN elements. After that we will describe how the editor has been specified using GMF and take a look at the customizations that had to be made and the model validation. Finally we will discuss some issues that are not yet fully decided.

6.1 Project Structure

The editor consists of several interdependent plugins (see figure 6.1). In this section we will have a look on their purpose and give a short description for each of them.

- **VSDT** This plugin is holding the model implementation as well as the various EMF and GMF model files. This plugin is at the lowest level of the plugin hierarchy. It does in fact provide all that is necessary for programmatically creating, editing and saving BPMN diagrams.

- **VSDT.editor** This plugin is holding the fully generated EMF tree editor. This editor is still necessary for creating and editing some of the non-graphical elements of the models. In the future this editor should be integrated together with the graphical part in a multi page editor.
Figure 6.1: Editor-Plugin Interdependencies. Components at the top of the figure require those at the bottom (transitively).

- **VSDT.edit** This plugin is holding the property item providers used by both the EMF and the GMF editor. The property item providers were first generated by EMF and then customized, e.g. for filtering some of the choices and for enabling property wrappers.

- **VSDT.diagram** This is the main part of the graphical editor.

- **VSDT.figures** This plugin is providing some highly customized helper classes, for instance the figure classes and abstract super classes holding a great deal of the customizations that were needed for the editor parts. They have been removed from the **VSDT.diagram** plugin for simplifying the regeneration of the plugin.

- **VSDT.custom** This plugin is not required for the execution of the editor. However, it provides some custom actions improving the editor’s usability and accelerating the execution of some frequently needed editing operations.

Since the editor is based on the Graphical Modeling Framework, the GMF plugin is required for the execution as well as all requirements of the GMF plugin.

Eclipse plugins may not have cyclic dependencies. Thus it is necessary to have two customization plugins. The first, **VSDT.figures**, is holding those classes that do not have requirements from the main plugin, **VSDT.diagram**, but which are required by the main plugin. The second, **VSDT.custom** is holding those classes that have dependencies from the main plugin but which are not required by the main plugin. Classes that both have requirements from and are required by the main plugin can not be sourced out of the main plugin.

It might seem reasonable to merge some of the plugins. But remember, that sometimes it is necessary to recreate the diagram plugin from scratch, e.g. when changing to a new major release of GMF. In such a case it will be advantageous not to have the generated classes mixed up with the custom ones.
6.2 The Domain Model

The BPMN specification is focused on the element’s graphical representation, neglecting a formal definition of the data format. It does not provide a XSD file, a class diagram or similar. Instead it only states which attributes the several elements need to have and what these attributes stand for. The rest is left up to the developers of the modeling tools. Thus it is nearly impossible to share BPMN diagrams between editors from different vendors.

The Ecore model has been created with the primary goal to be as close to the specification as possible. Every attribute used in the specification has its representation in the model. The inheritance relationships were derived from the element’s attribute lists, which often were said to “extend” some other attribute list. However, some of the classes were altered slightly in order to improve the usability of the editor. Figure 6.2 is showing only an extract of the BPMN model since the complete model would be far too large to be shown on a single diagram. The various supporting types are left out as well as the several attribute sets.

Figure 6.2: UML Class Diagram of the editor’s domain model (extract). Red: diagram root element; blue: nodes; green: connections; yellow: non-graphical supporting types; orange: attribute sets; white: abstract superclasses

6.2.1 Containment

Another problem in creating an Ecore model from the specification was that in an EMF model each element has to be contained in some other element (or, in case of the root element, in the resource). Whether a reference on an element should be a containment reference is depending on whether the element is “owned” by the
containing element. In most cases this is clear, like for Lanes being contained in Pools.

For graphical elements the containment structure is constrained by the GMF framework: A node may be contained in a diagram compartment only if the node’s domain element is contained in the compartment node’s domain element. It would be possible to alter the code in such a way that elements are shown in a different compartment than they are actually contained in, but this is not supported by the GMF tooling, making the generative approach a burden instead of an ease.

An example for this is the case of Processes and Lanes, which has been an open issue for some time. In the specification all the Flow Objects are contained in the Pool’s Process but shall be drawn in the Lane’s compartment. Thus the containment reference had to be changed so that the Flow Objects are contained in the Lanes. The Process still has a derived reference on the Flow Objects of all the Pool’s Lanes.

In other cases, especially for the non-graphical Supporting Types, the containment is not that clear. Messages are referenced by Activities and Events of some type and by Message Flows, whereas both the sending and receiving Flow Object as well as the (optional) Message Flow connecting both of them reference the same Message. Now where should the Message be contained? In this implementation the Message is contained in the Message Flow, if any, or else directly in the Business Process Diagram, the top-level element.

### 6.2.2 Attributes

Another issue were the numerous sub-type-specific attributes. Activities, Events and Gateways have a specific sub type, e.g. **Send**, **Timer** or **XOR**. All of these sub types require a different set of attributes. Basically there are at least three different ways to provide these attributes:

1. create several classes, one for each sub type
2. provide all the attributes in the base type’s class
3. provide the attributes in separate classes, being referenced by the base type’s class

Using the first pattern it would be very complicated to change an elements sub type after creation. In the second pattern, most of the attributes would remain unused and additional audits would be necessary for checking whether every attribute needed for the given sub type is set. In the third pattern the attributes are held in a non-graphical element which has to be synchronized with the element’s **type** attribute. The first pattern seems unusable to us. The second and third both have their pros and cons. For now we decided to use the third pattern. Since changing from the third to the second seems a lot easier than the other way round.

Similar to this case there are still some open issues concerning the model. Please refer to section 6.4 for more.
6.3 The Editor Specification

We decided to use the GMF framework for the implementation of the editor, which has been introduced in 3.1.2. Thus the basic editor could be designed easily by setting up a set of models.

However, using GMF brings some restrictions: In order to gain advantage from the generative approach one has to follow the structure and design patterns used by the GMF tooling, making it complicated to customize the editor by hand, since the developers can not transfer all of their expertise from GEF to GMF. Then again, using GMF all the code of each project will be more uniform than code handwritten by several different developers. Thus editors generated with GMF are very well suited for being maintained by large, diverging groups, since everyone familiar with GMF can understand the code.

As said before GMF makes use of a set of models: The Domain Model, the Graphical Definition Model, the Tooling Model, the Mapping Model and the Generator Model.

The Domain model used for the Visual Service Design Tool has already been introduced. The Graphical Definition Model for the VSDT, in which the various Nodes and Figures are defined, is making use of a number of custom figures, defined in another plugin. This was necessary because the figures have to be sensitive of the element’s state (e.g. its subtype). The Tooling Model consists of only a small number of palette entries. Instead of creating one palette entry for each element or even for each subtype as it can be seen in some other tools we decided to keep the palette clear and concise. The Create Event icon is used for Start, End and Intermediate Events, prompting the user which one should be created. Doing so we
hope for a good usability. A visualization of the Mapping Model, which glues the other models together and is then used to create the Generator Model, can be seen in figure 6.3.

The Business Process Diagram element has been declared as the root element. The only top level nodes are the three types of Artifacts and the Pools, which again can hold a number of Lanes. The Lanes finally hold the various Flow Objects. One of these, the Activity, is a container, too, and can again hold every Flow Object in its Subprocess Compartment, referencing the nodes defined before. Additionally the Activity Node has a second compartment for holding event handlers, that is, Intermediate Events attached to the Activity’s boundary.

Finally, the Generator Model has undergone only minor adjustments, e.g. concerning the file extensions or some layout styles, before it was used for generating the editor code.

6.3.1 Customizations

While the GMF tooling can create a readily usable editor with numerous additional features the generation is still not perfect. Using GMF generated code can be merged with that written by hand, thus bugfixes and missing features can be added to the editor without the risk of nullifying prior changes.

In this section we will take a look on the several bug fixes and improvements that have been added to the editor.

- On each child Edit Part (that is, Edit Parts which’s nodes are not drawn directly on the canvas) a extended GraphicalNodeEditPolicy has to be installed. This is needed to avoid the duplication of edges drawn between two nested nodes, which is a well-known bug in GMF[1] that has been scheduled to be fixed in GMF 2.0 (final). Until then this bugfix is absolutely necessary.

- When deleting a node from the diagram all the edges going in and out of the node have to be deleted, too. By default, however, only the edge’s visualizations are removed from the diagram, while the Connecting Objects remain in the model. To fix this the VsdtBaseEditHelper has to implement getDestroyDependentsCommand, collecting the connections to be deleted.

- To support the reorienting of connections, the connections’ Edit Helpers have to implement the getReorientRelationshipCommand method.

- Each of the Item Providers inherit from an extended ItemProviderAdapter, providing the Item Providers with special ItemPropertyDescriptors, wrapping EObject property values. This way referenced model elements can be expanded in the property view. Otherwise the user would have to use the EMF editor whenever he’d want to edit some non-graphical element’s values.

- The ModelingAssistantProvider has been customized to enable connection handles and to rearrange the elements in the popup menus.

1[see https://bugs.eclipse.org/bugs/show_bug.cgi?id=148021]
• The Edit Parts extend some abstract Edit Part holding most of the customizations. This way the customizations are clearly arranged and in some cases can be reused for more than one Edit Part.

• Each of the figures is heavily customized. This is necessary since most figures have to adapt to the underlying model element’s sub type and other attributes, enabling and disabling various markers on the figure.

• The validation audits (see section 6.3.2) have been implemented in Java. To avoid conflicts or unwanted dispose of implementations when the diagram editor is regenerated they have been moved to a separate class outside of the constantly regenerated main project.

• The VSDT.custom project contains some custom actions. These actions can be reached via the editor’s context menus and provide some shortcuts for editing operations that would otherwise be very time consuming or error prone. However, this project is not needed for the editor to work.

Beside the diagram editor the model implementation has been enriched with many additional methods. We will not enumerate every single change here but only state the categories of additions.

• Some customizations are used to instantiate non-graphical elements. For instance when an Activity’s type is set to a new value an Attribute Set for that type will be created and bound to the Activity. Further Expressions and IDs are instantiated on the creation of their parent element. Otherwise the EMF tree editor would have to be used for creating those elements.

• Some custom methods are frequently needed shortcuts, e.g. for getting a Flow Object’s Process by going all the way up to the Pool and from there to the Process, or to determine whether some element is the first in a Sequence or if an Intermediate Event is in normal flow.

• Custom methods are used for the synchronization of associated model elements. For example whenever an outgoing Sequence Flow is connected to a Gateway a new instance of the non-graphical element Gate is created, associated with the Sequence Flow and added to the Gateway (analogue when the Sequence Flow is removed or redirected).

• The custom actions mentioned earlier are using custom undoable methods in the model implementation, for instance for initializing a Pool’s Participant and other editing operations that can be automated. Due to the EMF framework these methods can be undone with no need for additional undo and redo methods.

While some of these customizations, like the custom actions, are intended to improve the editor’s usability, most are needed to avoid unwanted behavior in certain situations or for the further adjustment of the editor to BPMN, for instance by providing a number of additional audits, which will be the topic of the next section.
6.3.2 Validation

As mentioned before, the GMF provides an easy way to validate models created with the editor. This way illegal models and model elements can be identified. The validation consists of two parts.

- basic validation of multiplicities provided by EMFT Validation
- custom GMF validation

Using the EMFT Validation Project the editor is capable of validating the various model elements to conform to the minimum and maximum multiplicities given in the Ecore model. This is very useful since it excuses the programmer from implementing this simple but extensive validation by hand.

In cases a multiplicity is dependent on another attribute or another aspect of the element’s context, for instance when either attribute a or attribute b is required (or allowed) to have a value, but not both, a custom audit can be defined in the GMF mapping model. These audits are then tested automatically for each instance of the type specified in the audit’s definition.

Both the Object Constraint Language (OCL) \[25\] and Java can be used as validation language and attributes can be validated using regular expressions, too. In this implementation in most cases Java has been chosen as validation language since it does provide much better debugging and should be faster, too, since those checks do not have to be interpreted by an OCL parser first. On the downside Java constraints have to be implemented in the generated code while OCL constraints can be specified directly in the mapping model.

Currently there are about 60 audits used in the editor. The audits were derived from constraint that are given throughout the BPMN specification. The Java methods representing the audits are named as “[modelElement][nr][description]”. For instance, the audit Boolean start4messageFlowHasMessageTrigger(Start self) is the fourth audit for the Start Event element; it checks whether a Start Event with an incoming Message Flow has a trigger of type Message and returns TRUE or FALSE accordingly.

The model can be validated on-the-fly or on demand. If there are any errors or other problems found the node or edge corresponding to the erroneous model element as well as the resource holding the element can be given an error marker. The error is listed in the Eclipse Problem View and the description is given as a tool tip when hovering the mouse over the error marker.

6.4 Open Issues

As mentioned earlier there are still some open issues, especially concerning the model. Each of the alternatives have their pros and cons. However, the final decision on these topics has been postponed for now.
6.4.1 Attribute Sets

Depending on its type each Flow Object – Events, Activities and Gateways – needs different attributes. These are the possibilities to handle these type specific attributes:

- Define a separate class for each of these subtypes. This has the disadvantage that there will be many different classes – especially for the Events with their second subtype, Start, End and Intermediate. Furthermore it would not be possible to change a node’s type afterwards and it would require a lot of code duplication.

- Put all these attributes in the base class. This approach would require additional audits to decide if all the attributes required for the given type are set, since the attribute’s multiplicity is depending on the subtype and can not be validated by the built-in EMF validation feature. Further the property view would have to be configured to show only those attributes relevant for the subtype, which is currently not provided by GMF.

- Create a class for each set of attributes, being referenced by the base type’s class. Thus it is possible to use the built-in multiplicity validator while preserving the ability to change the subtype. In the property view the AttributeSet – as well as every other EReference – can be wrapped, so it can be expanded and the attributes can be set in the graphical editor, although they are held in a non-graphical element (see figure 6.4). On the downside the AttributeSet has to be synchronized with the type attribute, which can be done using the EditPart’s notifyChanged method or the model element’s implementation’s setType method.

![Figure 6.4: Property View with expanded Attribute Set wrapper.]

Currently the AttributeSet approach is being used. It might seem to be redundant using the type attribute as well as the AttributeSets. But other than the AttributeSet to be used the type can be selected from a drop down list in the property view, which will then initialize the AttributeSet.

Whenever reading one of the subtype’s attributes the referenced AttributeSet has to be casted to its real type. This approach does require some code duplication, since some subtypes, like Send and Receive Tasks, have common attributes. This could be improved by introducing additional AttributeSets holding the common attributes acting as a super class for the actual AttributeSets.

Despite the fact that it would require a number of additional audits, putting all the attributes in the base class might be reasonable, too. There would be no need for class castings, making the code much more readable. However, this is not practicable
until finding a way to dynamically filter the editor’s property sources according to the sub type. The generated property item providers follow the Singleton pattern, so there is only one instance of the item provider for each element of the same type. Providing a dynamically changing property provider would require serious customization of the editor's generated structures.

6.4.2 Expressions

Numerous BPMN element types have one or more attributes of type Expression. In the specification an Expression is not more than a wrapper for a String, holding the actual expression. We are not yet fully decided how to handle the Expression type.

- Keep close to the specification and use the Expression type. This way the Expression class could provide methods for validating the expression, or expressions could even be edited in a separate editor, providing features like syntax highlighting.

- Dissolve the Expression type and use simple Strings instead.

None of the advantages of using a special Expression class are currently used. The expressions can not be validated or evaluated, since the expression language to be used is unknown at the editor’s design time. Of course one could arbitrarily provide validators for several expression languages, like Java, JADL or XPath. Maybe these validator could even be provided together with an export wizard requiring expressions to be written in that language.

However, by now all these are not more than ideas.

On the downside using the Expression type is currently complicating entering expressions: The Expression type has to be expanded to edit the expression in the Expression. Using plain Strings the expressions could be easier edited in the property view using the standard String property provider.

So it might be reasonable to dissolve the Expression type and use simple Strings for now. However, dissolving the Expression type will be much easier than restoring it. Thus we will keep it for now.

Of course the best way would be finding a way to use the standard String property provider to edit the expression String wrapped in the Expression instance without expanding it in the property view.

6.4.3 Tasks and Subprocesses

We decided to use the type Activity for both Tasks and Subprocesses, so the user is free to expand a Task to a Subprocess. However, in the BPMN specification as well as in many other BPMN editors individual graphical elements are used for Tasks and Subprocesses.

- Represent both Tasks and Subprocesses with the Activity type. This has the advantage that the user can easily change from Task to Subprocess without deleting and recreating the element.
• Use different types for Tasks and Subprocesses. This is closer to the specification and has the advantage that the user is not confused with attributes relevant for Subprocesses only when dealing with Tasks and vice versa.

The use of the Activity type for both Tasks and Subprocesses has some clear advantages, like being able to switch from some Task type to a Subprocess type. However, using separate types for Tasks and Subprocesses would result in better icons for both. Using the same figure for both means that Tasks have a Subprocess compartment, too, making the figure unnecessarily larger. It is possible to reduce the compartment’s size when the type is different from Embedded, which is the only activity type that actually used the compartment. But this demands the user to set the Activity’s type to Embedded before inserting any child elements, and when changing the type to something else than Embedded the Subprocess compartment might be minimized with the elements still inside of it. The user would have to expand it again to remove those elements.

Using only one type and figure for both Tasks and Subprocesses seems be the best choice, but the Activity figure might need some improvement to fit all the needs.

In this chapter the BPMN editor that is part of the Visual Service Design Tool has been introduced. We have explained the domain model, how GMF was used for the editor’s generation and how it was extended with a number of custom features to improve its usability.

The next chapter will introduce the implementation of the transformation tool used for the mapping from BPMN to JIAC and BPEL.
7. Implementation: The Mapping

In this chapter we will describe the implementation of the model transformations from BPMN to BPEL and JIAC.

The methodology for both transformations is very similar. The mappings have been realized using the EMT framework, which has been introduced in section 4.2. However, the EMT framework has been used without its visual editor, which still seemed to be too unstable. Instead the framework has been optimized for manual use, meaning that with no big effort the rules can be directly implemented in Java instead of using the EMT compiler.

After giving an overview of the project and plugin structure we will introduce the Transformation Base, that is, the modified EMT framework, and the codomain models. After that we will explain in detail the stages of the mapping, from the normalization to the clean up, including the several rules designed to transform the rather unstructured BPMN graphs to structured BPEL and JIAC models. Finally we will discuss the remaining open issues.

7.1 Project Structure

Like the editor feature, the mapping features are spread across several interdependent plugins, as shown in figure 7.1. Again we will give a short description of the plugins. Note that every export feature may provide its own plugins which are not required to follow this structure used for the export features in this diploma thesis.

- **TransformationBase** This plugin is holding the classes needed for the rule based transformation. These are the slightly modified and optimized classes generated by the EMT framework. This plugin can be used for any EMF model transformation, independently of the source and target language.

- **BpmnExport.refmodel** The reference model connecting the source model with the target model. The reference model is independent of the target language and can be used for the transformation to both BPEL and JIAC.
7. Implementation: The Mapping

Figure 7.1: Mapping-Plugin Interdependencies. Components at the top of the figure require those at the bottom (transitively). Different colors mark a possible composition in Features.

- **BpmnExport.rules** This plugin is holding those parts of the rules that are independent of the target language. It provides the normalization rules, which operate on the source model only, as well as the wrapper classes for the rules of the structure mapping, which provide the patterns for the pattern matching and are independent of the target language, too.

- **BPEL** This plugin is holding the BPEL metamodel.

- **BPEL.edit, BPEL.editor** These plugins are required for providing a standard EMF editor for the BPEL models. They are not needed for the export functionality but may be useful for easier validation and editing of the resulting model.

- **Bpmn2Bpel.export, Bpmn2Jiac.export** These plugins are holding the logic needed for the export to BPEL and JIAC, including the export wizard, a visitor and the remaining rules. The JIAC export plugin also contains the libraries holding the JIAC IV metamodel and code generation routines, which are realized using JAXB.

Note that neither the TransformationBase nor the .export plugin do require any of the EMT plugins. All classes needed for the application of the rules are contained in the TransformationBase plugin and no other plugins than those already required for the editor are needed.

### 7.2 The Transformation Base

The TransformationBase plugin is based on the EMT framework introduced in section 4.2. The plugin consists of only 11 classes. The Transformation class is aggregating a set of rules and tries to execute them as long as possible. The AbstractRule class is providing the methods needed for the rules while the AbstractWrapper class is the superclass for “wrappers”, which define the rule’s left hand side (LHS) and negative application conditions (NACs) and are used for the pattern matching, which is done by the Matchfinder class. For each object in a rule’s LHS a Variable is defined. A number of Queries are assigned to the Variables which have to be fulfilled in the course of the pattern matching algorithm so that the rule can be applied.
7.3 The Codomain Models

These classes were first generated by the EMT compiler and have then been optimized with respect to easier maintainability and performance. Some minor bugs have been fixed and the backtracking algorithm has been optimized, too, making it more efficient. A number of sub routines have been written, simplifying the creation of Variables and Queries.

The generated rule implementations had to use EMF’s reflective API. For creating a instance of a EClass (that is, a EObject) the class had to be searched for by its name in each EPackage that has been assigned to the transformation model. Obviously this is not very performant, hard to maintain and error prone – what if there are two EClasses with the same name? This is not needed anymore when writing the rules by hand, so the resulting rule classes are relatively slim while still using the benefits of the transformation framework, for instance the fully implemented pattern matching algorithms.

Another benefit of not using the graphical rule editor is not being bound to its restrictions. A “classical” rule has a Left Hand Side (LHS, the pattern), some Negative Application Conditions (NAC), conditions and one Right Hand Side (RHS, the result). This is not very advantageous in the case of BPMN, since most elements of BPMN have several sub types, which have great influence on the mapping. The differences between the several sub types can not be expressed in a rule’s single RHS, so for each subtype a new rule would need to be created. Let alone all the possibilities of containing elements for each of these types! Using the classical LHS-RHS approach would have resulted in literally *hundreds* of rules.

Thus the most important change is the introduction of the `Rule.apply()` method, replacing the existing methods for building the rule’s RHS. The old methods were subdivided into creating, changing and removing objects in the RHS. Having only one single method for creating, changing and deleting objects in the course of the rule’s execution, the readability and maintainability of the rule classes is increased a lot and new elements, such as conditionals and loops, can be introduced.

7.3 The Codomain Models

This section will outline the metamodels for the target languages.

**BPEL and WSDL**

The EMF Ecore models for the Business Process Execution Language (BPEL) and Web Service Definition Language (WSDL) were created from the BPEL XML Schema Definition (XSD) [3, Appendix D]. Thus, the model’s XML code is equal to the original BPEL XML code and no transformation from one to the other is needed.

**JIAC**

The JIAC domain model was taken without modifications from the existing JIAC IV projects. Along with the model all the code generation routines could be reused.

Unfortunately the existing JIAC metamodel is implemented in JAXB[1] This has the consequence that JIAC model elements can not be used in the EMT pattern

56 7. Implementation: The Mapping

They can be created in the course of the rule’s application, though. This has no impact on the actual mapping, since in these stages only the source model and the reference model are needed for the pattern matching. But with this restriction it is not possible to write clean up / beautifier rules for JIAC using the EMT framework.

The next version of JIAC, JIAC TNG, will be based on EMF. As soon as JIAC TNG is available the transformation should be converted to JIAC TNG, including a beautifier.

The Reference Model

For the rule based transformation a minimal reference model has been developed (see figure 7.2). This model’s purpose is to hold the source and target models together. The Mapping Root is holding references to the BPMN root element, namely the Business Process Diagram, and the target language’s top-level structures. Further it holds a list of References, which subdivide in Activity References and Sequence References. These are needed to connect each element of the source model with its representation in the target model. Additionally they mark an element as being processed simply by referencing it. This can be used as a pattern in a Negative Application Condition (NAC) in the transformation rules to prevent the rule from being executed twice for the same element. While Activity References permanently connect each Activity of the source model with its counterpart in the target model, Sequence References, which are used to connect sequences to be created in the target model with the first and the last node belonging to that sequence in the source model, are reduces in the course of the transformation.

Figure 7.2: The Reference Model. The MappingRoot element is containing both the source and target model.

The reference model is independent of the source and target model. While the Reference types are tailored to fit workflow models all the references to the source and target model elements have the type Object and thus can be used with any metamodel.

7.4 Stages of the Mapping

The transformation has been subdivided into several stages which are finished one after another. Some of these stages are identical for each mapping, independent of the target language. Others are similar in their structure but have to be implemented individually for each target language.
7.4. Stages of the Mapping

1. **Normalization**: Puts the model in a uniform form to facilitate the later stages.

2. **Element Mapping**: Transformation of the several activities to a unsorted collection of concurrent sequences.

3. **Structure Mapping**: Mapping of control flow, combining the previously created sequences in larger structures.

4. **Cleaning Up**: Cleaning up the resulting workflow.

The central component in the transformation is the *Visitor*. It extends the reference model’s MappingRoot class and thus holds the References connecting the source and target models. At first it calls the pre transformation, that is the *Normalization Stage*, then it starts its top-down traversal, the *Element Mapping*. After that the post transformation is called, which consists of the *Structure Mapping* and the *Clean Up*. Finally the results are saved in the directory that has been specified by the user. In the following sections we will explain the several stages in detail.

### 7.4.1 Normalization

This is the first stage of the transformation and a internal transformation of the source model. Thus this stage is completely independent of the target language. This stage is intended to put the model in a uniform structure to facilitate the following stages of the transformation. The model will not be saved after the transformation, thus the changes done here won’t affect the model file.

Each of the normalization rules presented here are very short and concise. Often the LHS consists of only one or two elements. Thus it does pay of highly if the number and complexity of rules needed for the Structure Mapping can be reduced by introducing these normalization rules. In the following paragraphs we will introduce some of the normalization rules and give a high level view on the diagram before and after the application of the rule.

**Split Gateway Rule**

With this rule Gateways with both multiple incoming and outgoing Sequence Flows are split in two Gateways, one with multiple incoming, the other with multiple outgoing Sequence Flows (see figure 7.3). This rule is necessary for facilitating the Block Rule, which will be covered later on in this chapter. After the application of this rule each Gateway is either a forking Gateway or a merging Gateway.

![Figure 7.3: The Split Gateway Rule.](image)
Insert Gateway Rule

With this rule an Activity with multiple incoming and/or outgoing Sequence Flows is split up in the Activity alongside with one or two Gateways (see figure 7.4). This rule is necessary to facilitate the Sequence Rule and the Block Rule. After the application of this rule each Activity will have at most one incoming and outgoing Sequence Flow.

![Figure 7.4: The Insert Gateway Rule for the case of multiple outgoing Sequence Flows.](image)

Insert Empty Rule

With this rule two Gateways that are directly connected by a Sequence Flow are separated by an Activity with task type None (see figure 7.5). The Activity will be mapped to a no-op activity in the target language and can eventually be removed in the Clean Up layer. This rule is necessary to facilitate the Block Rule and Loop Rule. After the application of this rule there will be no directly connected Gateways anymore, meaning that there are not empty sequences between Gateways and that every sequence spanning from one Gateway to another is beginning and ending with a Activity with only one incoming and outgoing Sequence Flow.

![Figure 7.5: The Insert Empty Rule.](image)

Example

Figure 7.6 is showing a short example of how the Normalization rules help to change a loop structure drawn in a very short and easily understandable way to a more extensive representation with exactly the same semantics that can be mapped by the Loop Rule, being introduced in the Structure Mapping.

Processing of Unstructured Workflows

Another part of the normalization mapping would be the structuring of unstructured workflows, as introduced in section 3.2.1. This part of the mapping has not yet been implemented in this work, thus only workflows that are already in a structured form – apart from the minor defects stated above – can be mapped. Some more research will be needed before this part of the mapping can be completed and a number of additional rules will be necessary to cover each possibility of unstructured workflows.
7.4.2 Element Mapping

The second stage of the mapping, the Element Mapping, is the only stage that is not rule based. Instead the mapping of the several BPMN elements to their correspondents in BPEL and JIAC has been implemented using a visitor-like top-down traversal of the source model. A rule based transformation would have been possible, too, but not necessary in this case, and a set of rules would be much harder to debug and to maintain. Further, the top-down traversal is faster than the pattern matching required for a rule based transformation.

With this stage of the mapping two new models will be build alongside the source model: The reference model and the target model. While in some other transformations the source model often is annotated and reduced, the source model will stay intact in our case. Instead the reference model is used to connect it with the target model to be created. This has the advantage that the whole source model is still available for checks and rule patterns later on.

For each element of the source model – that is, mainly the Pools and Flow Objects – a corresponding element of the target language will be created and connected to the source element with Reference objects. However, these elements are only loosely gathered in a parallel structure. The right execution order, alternative paths and concurrency will be added later, in the Structure Mapping.

The visitor is starting at the root element, the Business Process Diagram. One program module, that is, a BPEL process or a JADL planelement, is created for each Pool in the diagram. For the Pool as a whole a parallel structure in created, the Pool’s elements are traversed and the corresponding elements of the target language are created. If no mapping is defined for some element than a no-operation element is created with a short description, for instance the original element’s name. In BPEL this will be an empty activity, in JIAC a logwarn. For each mapping an Activity Reference is created and referenced by the Mapping Root. These objects are holding references to the mapped element and its counterpart in the target language and are needed for some of the transformations in the Structure Mapping.

In BPMN each Flow Object can have Assignments, which do not have a graphical representation in the diagram. These Assignments are mapped to assignments in the target language, e.g. an assign in BPEL or a bind in JIAC. These assignments,
together with the Flow Object’s mapping, are wrapped in a sequence element. For this sequence a *Sequence Reference* is created, which is needed for the next stage, the Structure Mapping, and is registered at the Mapping Root object\(^2\).

Finally the sequence is inserted into the parallel structure together with the sequences that have been created for the other Flow Objects in the same (Sub-) Process. Later, in the Structure Mapping, these sequences will be combined to larger sequences, decision structures and loops. Sequences, that can not be combined, remain concurrent in the parallel structure. Figure 7.7 should illustrate the concept.

![Diagram of mapping sequences](image)

Figure 7.7: Using sequences to wrap mappings of FlowObjects and Assignments in the Element Mapping. Yellow: Source Model, Green: Reference Model, Red: Target Model.

### 7.4.3 Structure Mapping

A great challenge consisted in mapping the BPMN diagram graphs to the structured executable languages. While BPMN diagrams are directed graphs, BPEL and JADL are block oriented. This means that each loop or block structure has to have exactly one entry and one exit point. It is not possible to “break out” of the block or to alternate between two concurring blocks.

Mapping a directed graph to an equivalent block structure consists of two steps: Firstly one has to alter the diagram in an endogenous transformation so it will be in a block structure, then the exogenous transformation to the target language can begin.

The first step should be done in the Normalization stage of the mapping, so that new elements can be regarded in the Element Mapping. So everything that’s left at this point is to identify graph structures that can be mapped to block structures.

As written earlier in section 3.2.1 a structured workflow model basically consists of only three elements: Sequences, choice/parallel blocks and loops. In BPMN there are various possibilities of realizing these structures [1, 30]. Still three rules are enough to cover the biggest part of the Structure Mapping.

1. **Sequence Rule**: Those sequences in the target model corresponding to Flow Objects connected with a Sequence Flow are combined to larger sequences.

---

\(^{2}\)which actually is the Visitor itself
2. **Block Rule**: Blocks in the BPMN diagram are identified and a corresponding block structure is created in the target model.

3. **Loop Rule**: Loops are identified and a loop structure is created. The until- and the while-parts of the loop are put in the structure.

Note that these three rules are still not enough to do the whole Structure Mapping. Some more rules are needed for special cases like open blocks, that is, blocks that are defined by only the forking or the merging Gateway. These blocks are essential for defining workflows with multiple Start or End Events. Further BPMN’s exception handling does require some additional rules. All these rules are basically very similar to the three rules presented here and will not be explained in detail.

The figures 7.8, 7.9, and 7.10 will roughly illustrate the three main rules. The yellow nodes are elements of the source model, BPMN. The red nodes are elements of the target model, BPEL or JIAC. The green nodes are elements of the reference model, connecting the source and target model. The area within the dotted rounded rectangle is the pattern that has to be matched in the source model.

**Sequence Rule**

In this rule, shown in figure 7.8, sequences of sequences are merged to larger sequences. It is necessary that in the Element Mapping each resulting element of the target language has been initially wrapped in a sequence and referenced by a Sequence Reference. If this has been done each of these elements will be both first and last and the Flow Object’s incoming and outgoing Sequence Flows will be the incoming and outgoing of that Sequence Reference.

The rule engine will look for occurrences of Sequence Flows being outgoing of one and incoming of another Sequence Reference. If this is the case the sequences referenced by the Sequence References are merged, preserving the order of the child elements. The first Sequence Reference’s first and incoming elements will be the first and incoming of the new sequence and the second Sequence Reference’s last
and **outgoing** elements will be the **last** and **outgoing** elements of the new sequence. Finally the second sequence, together with its Sequence Reference, can be removed from the model.

**Block Rule**

The **Block Rule**, shown in figure 7.9, will identify blocks defined by a forking and a merging Gateway and create a new block structure corresponding to the type of the forking Gateway. The type of the merging Gateway can not be taken into account yet. After the block structure has been created all sequences starting at the forking Gateway and ending at the merging Gateway will be put into the block structure. Finally the sequence holding the forking Gateway and the sequence holding the merging Gateway will be combined similar to the **Sequence Rule**, so that the block can then be integrated into a larger sequence.

![Figure 7.9: The Block Rule](image)

This rule requires that there is more than one sequence in between the Gateways and that there are no sequences starting at the merging Gateway and *not* ending at the forking Gateway and vice versa.

**Loop Rule**

The **Loop Rule**, shown in figure 7.10, is basically very similar to the **Block Rule**. The main difference is that there must not be an arbitrary number of sequences from the first to the second Gateway but exactly one sequence from the first to the second and exactly one sequence from the second to the first. Further, the first Gateway has to have one more incoming Sequence Flow and the second Gateway one more outgoing Sequence Flow, that is, the entry to and the exit from the loop. As shown earlier in figure 7.6 after the Normalization stage both *while* and *until* loops will have this form and thus are covered by this rule.
7.4. Stages of the Mapping

When executed, the sequence from the first to the second Gateway (the until-part) will be copied and inserted before the loop structure, since it has to be executed at least once. The sequence from the second Gateway back to the first Gateway (the while part) will be part of the loop's body, together with the copy of the first sequence. The loop condition will be taken from one of the Sequence Flows going out of the second Gateway, depending on which Sequence Flow's condition is a Expression and which is the default.

7.4.4 Cleaning Up

This last stage of the mapping is intended to clean up the target model of redundancies and defects that may have arisen during the mapping.

This list is showing some possible actions for the clean up stage:

- remove no-op activities resulting from the Activities inserted in the Normalization stage
- resolve unnecessary nesting, for instance sequences nested in sequences
- resolve singleton sequences

Remember that the Clean Up stage could not be implemented for JIAC IV, since the existing JIAC metamodel is implemented using JAXB which is not compatible with EMT and thus can not be used in its rule patterns.
7.5 Open Issues

Like for the editor there are still some open issues concerning the implementation of the mapping.

7.5.1 The Mappings

Both the mapping to BPEL and to JIAC are not yet fully implemented. Some of the mappings given in the BPMN Specification are somewhat ambiguous or very difficult to implement, which is why they have been postponed for now. The mapping to JIAC is not fully specified yet and will need further refinement in some points. Currently the mapping is focused on the planelements, while the mapping to agents is at a very early stage.

Further, the current mapping is addressing JIAC IV, which will be replaced by JIAC TNG soon. The mapping will have to be changed then. Fortunately the metamodel for JIAC TNG will be based on EMF, so it will be easier to integrate in the rule based approach, including a working Clean Up stage.

7.5.2 Unstructured Workflows

The current rules do not yet support the transformation of unstructured workflows. The normalization of unstructured workflows, as described in section 3.2.1 will require a number of additional, elaborate rules, but we are confident that even those rules will be possible with the current approach.

The existing Block Rule and Loop Rule will have to be completed by taking the joining Gateway into account. Currently the type of the decision structure is determined only by the splitting Gateway: If the split is a AND-Gateway, a parallel structure is created, in the case of a XOR-Gateway an exclusive decision structure is created, and so on.

If the joining gateway is of a different type this can result in deadlocks and loss of synchronization, that is, multiple instances of the same activity being active at the same time. While it may be a good thing that it is not possible to accidentally cause a deadlock, multiple instances can be intended by the modeler. In this case, for instance when a splitting AND-Gateway is followed by a joining OR-Gateway, the remaining workflow after the Gateway would have to be encapsulated in a second process, which can then be called each time the workflow reaches the end of the decision structure.

Another situation, where splitting and joining Gateways of different type are useful, is that of a splitting AND-Gateway and a joining XOR-Gateway: In this case the workflow should be continued right after the first branch has completed, not waiting for the others. This could be achieved by the use of concurrent flow and auxiliary variables.

In this chapter we described how the transformation tool was implemented. We explained the several stages of the mapping and introduced the reader to the most important transformation rules.

The next chapter will give some examples to illustrate the transformation from BPMN to JIAC.
8. Examples

This chapter will illustrate the transformation from BPMN to JIAC by giving two examples. Firstly the several stages of the mapping will be visualized. Then a realistic example of modeling and generating a JIAC planelement will be given.

8.1 Stages of the Mapping

In this first example we will illustrate the several steps of the mapping with a small example. Figure 8.1 shows a simple workflow, mainly consisting of a loop structure.

![Figure 8.1: The BPMN-Diagram to be transformed](image)

In the following (figures 8.2 to 8.6) we will present the source model (yellow), the reference model (green) and the target model (red) for the initial BPMN diagram as well as after each stage of the mapping. Note that we will show only the most relevant parts of the instance models. Neither the wrapping Pool and Process nor non-graphical elements like for instance for the loop’s condition or the Activity References will be shown. Also all attributes as well as some references will be left out, e.g. the incoming and outgoing references from the Sequence References to the Sequence Flows. For the target model an abstract notation has been chosen, showing only the basic concepts, like activities, sequences and loops.

![Figure 8.2: The Instance diagram of the initial source model.](image)
Figure 8.3: After the Normalization Stage: Two additional Gateways and an Activity were added, along with the necessary Sequence Flows.

Figure 8.4: After the Element Mapping: Each element has been mapped to a activity of the target language and wrapped in a sequence. Sequence Reference are connecting the source and the target model.

Figure 8.5: After the Structure Mapping: The singleton sequences have been combined to larger sequences and a loop structure. Note that the sequence making up the until-part of the loop has been copied and inserted before the loop.
Figure 8.6: After Cleaning Up: The sequences have been flattened and the no-op activity resulting from the Activity inserted in the Normalization stage has been removed. The source model has been left out in this diagram.

8.2 Code Generation

In this example a simple BPMN diagram as shown in figure 8.7 will be transformed to JIAC.

Figure 8.7: RSS-Client Example

The diagram is showing a simple RSS client. After being started the client is connecting to a server and requesting the news feeds the user subscribed on. After having received the response a Script Task will build the GUI. If there are any new messages in the feed those messages will be parsed and displayed, which is done calling a separate action. Otherwise a message will be displayed.

Note that the resulting code displayed in the following listing is not yet executable. Variables have to be passed to the speechacts and conditions. Further, the actual logic for displaying the GUI, parsing the news data etc. has been left out and the plan element still has to be put in the context of a service and given a precondition and effect.
(act RSS_Client_act
  (pre true)
  (eff true)
  (script
   (par
    (seq
     (loginfo "Start"
      (Request_News)
      (Receive_Data)
      (loginfo "Initialize GUI"
       (branch (Any_News_conditional)
        (seq
         (seq
          (loop (Display_News_loopCondition)
            (Display_News)
            break
          )
        )
        (seq
         (loginfo "Display Message"
          )
        )
      )
    )
    (loginfo "End"
     )
   )
  )
 )
// End RSS_Client_act

(act Display_News
  (pre true)
  (eff true)
  (script
   (par
    (seq
     (loginfo "Parse News Data"
      (loginfo "Display Formatted News"
       )
    )
   )
  )
 )
// End Display_News

(cond Display_News_loopCondition
  (comp hasMoreNews )
  true
 )
// End cond Display_News_loopCondition

(cond Any_News_conditional
  (comp gotNews )
  true
 )
// End cond Any_News_conditional

(send Request_News
 )
// End send Request_News

(receive Receive_Data
 )
// End receive Receive_Data
9. Conclusion

In this work a Visual Service Design Tool for the model driven development of programs for the multi-agent system JIAC has been developed. The tool is capable of transforming diagrams given in the Business Process Modeling Notation to JIAC and BPEL.

Firstly a survey on various workflow and business process notations has been made which lead to the decision to use BPMN as the source language for the mapping. After that we stated some of the goals and benefits of model driven engineering, followed by an introduction of the Eclipse Modeling Framework and the Graphical Modeling Framework. Further some approaches and known problems in the field of workflow transformation have been discussed, at which the main problem turned out to be the transformation of unstructured workflows to structured ones. For that purpose we introduced the rule-based graph transformation framework EMT.

We developed a mapping from BPMN to JIAC IV, which is concentrating on the control flow while the organizational structures, that is, the agents and platforms, are left out yet. We have discussed a number of approaches how BPMN diagrams could be mapped to agent concepts which are not covered by the original specification.

The visual editor for creating the business process diagrams has been developed using Eclipse GMF. Therefore the BPMN specification had to be implemented as an EMF Ecore model and mapped to nodes matching the notations given in the specification. Further, a number of constraints have been extracted from the specification and implemented as a model validation feature.

The mapping has been implemented as a transformation tool using both a top-down traversal and a rule-based transformation. The rules have been realized using the EMT framework. While well-structured diagrams can be transformed nicely there is still some work to do to support the transformation of unstructured workflows. The transformation tool has been integrated in the editor as an export wizard. Along with the mapping to JIAC also the mapping to BPEL that is proposed in the BPMN specification has been implemented.
9.1 Future Work

Future work to be done includes the review of some details of the mapping given in this thesis as well as the improvement of the Visual Service Design Tool and its integration in the larger context of the JIAC Toolsuite.

9.1.1 Completion of the Mapping and JIAC TNG

The most urgent work to be done is the completion of the mapping specification with respect to agents and agent platforms, which are not regarded in the current mapping yet. A number of possible approaches for the mapping of BPMN elements to agents has been given in section 5.2 of which the introduction of new custom Artifacts representing agent concepts seemed to be the most promising one.

Further work has to be done extending the transformation rules to cover more, possibly all kinds of unstructured workflows.

Eventually the mapping will have to be ported to the upcoming JIAC TNG, replacing the current JIAC IV soon.

9.1.2 Import and Round-Tripping

Once the mapping from BPMN to JIAC is complete the direction from JIAC back to BPMN should be implemented. For this it would be favorable that both the mappings from BPMN to JIAC and from JIAC to BPMN were bijective. Otherwise it could not be granted that the consecutive execution of the mappings will results in the original diagram again, so that if $exp: BPMN \rightarrow JIAC$ and $imp: JIAC \rightarrow BPMN$ are the mappings then $imp \circ exp = id$ and $imp(exp(x)) = x$.

Further, the export and import features together could be used to provide round-trip engineering, so that changes on the generated JIAC programs can be transferred back to the BPMN diagrams. In the case that the mappings can not be bijective round-trip engineering still could be provided by extending the reference model.

9.1.3 Rich Client Platform and GMF 2.0

As mentioned earlier GMF can be used to generate not only plugins but also self-contained applications, so called Rich Client Platform (RCP) applications.

GMF is supporting the creation of RCP applications from version 2.0M4 on. The development platform, however, was GMF 2.0M2 which had no RCP support. Since the later milestone versions were not rid of all bugs we decided to wait for the GMF 2.0 final release scheduled for Friday June 29, 2007\footnote{http://wiki.eclipse.org/index.php/GMF_Project_Plan} instead of adapting the existing projects with each milestone release.

As soon as the final version of GMF 2.0 is available it would be reasonable to refactor or recreate the editor with this new version to benefit from all the new features. Also the Visual Service Design Tool should be created both as plugins and RCP application to reach as many interested parties as possible.
9.1.4 Integration in the JIAC Toolsuite

Finally the Visual Service Design Tool could become a part of the existing JIAC Toolsuite. Being integrated into the Toolipse environment, which is based on the Eclipse platform, too, the VSDT could contribute by providing easy, model driven generation of new JIAC services while at the same time it could benefit from the numerous features aiding the creation and maintenance of JIAC programs that are currently aggregated in the JIAC Toolsuite. Further, the resulting tool could be enriched with a repository feature for the organization and reutilization of existing business process diagrams.
A. User Guide

A.1 Introduction

The Visual Service Design Tool is an Eclipse based graphical editor for designing, validating and exporting Business Process Diagrams based on the Business Process Modeling Notation.

Features

- as a plugin for the popular Eclipse Framework the editor is platform independent, easy to install and has a familiar look and feel
- using the Eclipse Graphical Modeling Framework and a modular structure consisting of several sub-plugins the editor can easily be modified and enriched with new features, e.g. additional custom actions or export wizards
- it provides all the basic editing features like undo and redo for each editing operation, zooming, printing diagrams to various image formats, and many more
- the editor is based on the BPMN specification, which is easy to understand by business men and formal enough for being exported to executable languages
- the diagrams drawn with the editor can be validated according to the constraints given in the BPMN specification
- diagrams valid to the specification can be exported to executable languages, such as WS-BPEL and JIAC

Components and Dependencies

The Visual Service Design Tool consists of several features:

- **VSDT** This feature is holding the plugins for the visual editor.
- **BpmnExport** This feature is holding plugins needed for the transformation, like a reference model and a transformation system.

- **Bpmn2Bpel** This feature is holding the plugins needed for the export to BPEL, such as a wizard, transformation rules and a visitor

- **Bpmn2Jiac** This feature is holding the plugins needed for the export to JIAC IV, such as a wizard, transformation rules and a visitor

Since the editor is based on the Graphical Modeling Framework, the GMF PlugIn is required for the execution as well as all of its prerequisites.

**The Eclipse Environment**

This section will briefly introduce those parts of the Eclipse GUI that are relevant for the work with the Visual Service Design Tool.

![A snapshot of the Eclipse GUI.](image)

**The Project Explorer**

Here the user can manage his projects and create and delete files. Note that Eclipse provides different similar views for managing files, e.g. the Navigator or the Package Explorer. However, only the Project Explorer is supported by the GMF, which allows to expand a Business Process Diagram file in the Project Explorer and inspect it's elements without actually opening it.

**The Editor View**

The editor window is shown automatically when opening a file in the Project Explorer. Depending on the PlugIns currently installed this can be a plain text editor, a browser, an elaborate code editor or some sort of graphical editor. For the Visual Service Design Tool there are two editors available. The visual editor is
for drawing the business process diagrams figures and connections. Still there are non-graphical elements which can’t be represented with nodes and connections. For managing and editing those elements there is a tree editor which hierarchically lists all the elements.

**The Graphical Editor**  This editor is opened when the diagram file is clicked. On the right side there is a palette with all the nodes and connections. Not all of the nodes can be placed directly on the canvas, e.g. the various Flow Objects can only be placed within a lane, with again must be placed inside a pool.

**The Tree Editor**  The tree editor is sometimes needed for managing and editing those parts of the Business Process Diagram that do not have a graphical representation, e.g. Assignments, Participants, or Messages. Note that the tree editor is more powerful than the graphical editor, and the diagram might be invalidated when doing certain operations in the tree editor. Where ever possible the graphical editor should be used. The tabs at the bottom line of the editor provide some selection-sensitive views on the tree. You can select some element in the outline (see below) and the tabs will show e.g. this elements parents or children in different ways.

**The Properties View**

Although some attributes, like an element’s name, can be edited in the editor view as well, for most other attributes the properties view will be needed. Here all the attributes relevant for the user can be seen and edited. All changes done in the properties view can be undone and redone; the editor is updated immediately.

**The Outline**

This view provides a short outline of the current editor’s content. In case of a graphical editor this could be a miniature view of the entire diagram, in case of a tree editor an additional tree editor for easier navigation.

**The Problem View**

This view lists all the problems that eventually have been found in the model, subdivided in errors and warnings. By double-clicking one of the items the editor will focus on the element the problem occurred on.

**The Error Log**

Other than the Problem View the Error Log will log problems that happened with the editor itself. So if you should encounter strange behavior or in case the editor should crash you can check here for the reason and send in an error report.
A.2 Basic Tutorial

In this section we will explain the basic steps how to create a simple Business Process Diagram.

1. First a **new Project** and a Diagram have to be created. To do so select  *New → Project → General → Project* in the menu bar and then  *New → Example → Vstd Diagram*. Open the Project Explorer to see the newly created Project and within the project two files. The first file is for the pure model data and the other one holds the layout information for the diagram.

   *Note* that both files are stored in XML format and can be edited with a text editor, too. However, you should do so only to fix a broken file. The diagram file can be recreated from the model file by right clicking it and choosing to initialize the diagram file.

2. **Open both files** by double clicking them. The model file will be opened with the tree editor, the diagram file with the graphical editor. For now, ignore the model file.

3. In the graphical editor select a **Pool** from the palette. Move the mouse to the canvas. Note that the mouse cursor changes where you can create a Pool. Push the mouse button and drag it to the lower right to create a large Pool. Enter a name for the Pool when you are prompted to.

4. Select the **Lane** element from the palette and click somewhere on the Pool. The Lane will be created in the top of the Pool, independent from where you click. According to the BPMN specification the first Lane will be invisible (faded out in the editor), so don’t be confused.

   Create a second Lane on the Pool and both Lanes will be visible. Note that the Lanes can not be resized or moved. They will adapt automatically to the elements within. You can right-click the Pool and select  *Auto Size* from the format menu and the Pool will automatically adapt it’s size to the Lane’s size, too.

5. Let’s create some **Flow Objects** inside the Lanes. Hover the mouse over one of the Lanes until you can see a miniature palette floating over the Lane. Select the Start Event icon and name it (see figure A.2). Create an Activity and an End Event. Try different ways, e.g. from the palette or by hovering the mouse.

   ![Figure A.2: Creating a Start Event](image)

Select the Sequence Flow icon from the palette and connect the Start Event with the Activity and the Activity with the End Event by pressing the mouse button on the source and dragging it to the target. When connecting the Activity be sure to aim for the label. If you hit the Activity’s compartment you can not create a connection. You can change the routing style from the toolbar or add bendpoints to a connection by dragging it.
6. Now select the **Message Flow** icon from the palette. Select the End Event as source and draw the Message Flow to some point beneath the Pool and select to create a new Pool element there (see figure A.3). Expand the Pool to the entire width of the diagram.

![Figure A.3: Creating a Message Flow to a new Pool](image)

Right-Click both Pools and select **Initialize Participant** from the BPMN menu. Take a look at the properties view to see the newly created Participant elements. Now right-click the Message Flow and select **Initialize Message** (see figure A.4). Note that the End Event’s type changed to **Message**. A new Message has been created and automatically associated with the Pool’s Participants, the Message Flow and the source and target (if possible).

![Figure A.4: Initializing the Message](image)

7. Let’s associate a Data Object with the Activity. Hover the mouse over the Activity and select the incoming Arrow. Drag the mouse to some point outside of the Pool and release the mouse button. Select to create an Association to a new Data Object element (see figure A.5). Select the Association and set the Direction Type to **To** in the properties view (if you dragged the outgoing arrow before select **From** instead).

![Figure A.5: Creating an Association to a new Data Object](image)

To complete this step select **BPMN → Initialize Input Set** from the Activity’s context menu. Notice the new Input Element in the Activity’s property sheet (see figure A.6). This Input Set references all the Activity’s incoming Data Objects.

8. Finally select **Validate** from the Diagram menu. You might notice some errors or warnings in your diagram. For now you can ignore these, but for exporting diagrams to executable code there should be no errors left in the diagram.

Save your diagram and change to the tree editor. Here you can inspect the attributes of non-graphical elements like the Message, the End Event’s Message Trigger Attribute Set or the Activity’s Input Set. By right-clicking the Business
Process Diagram element in the tree editor you can create new Supporting Types, e.g. for Messages and Implementations. Note that you might have to close and reopen the graphical editor for changes made in the tree editor to take effect.

A.3 How To . . .

...draw Flow Objects on the Canvas?

Although such diagrams can be seen in various papers, Flow Objects can not be drawn on the canvas. Instead, each Flow Object has to be contained in a Lane, and the Lane has to be in a Pool. However, one Pool per diagram can be set to have an invisible border (you can set more than one Pool to be invisible, but this will result in a validation error).

...draw a Flow Objects in a region of the Pool that is not covered by the Lane?

Lanes automatically adapt to their contents. If you want to draw a Flow Object in a place that is not covered by the Lane, for instance when starting an alternative path after a Gateway, you can stretch the Lane by inserting the element somewhere in the Lane and gradually moving it toward the Lane’s lower border.

...create an Intermediate Event on an Activity’s boundary?

Hover the mouse over the Activity. A miniature palette holding only an Intermediate Event will pop up. select the Intermediate Event and it will be created on the Activity’s boundary. You might have to select Auto Size from the Activity’s context menu if the Event is not placed properly. Select the Event and set its type as you like (see figure A.7).

...draw Artifacts inside of a Pool?

Contrary to Flow Objects, Artifacts can not be created inside of a Pool. However, you can create the Artifact on the canvas and drag it over the Pool. But remember that the Artifact is over, and not in, the Pool, so it will not be moved together with the Pool.
...enter a Time Date value?

This value is expected in quite complex form, because it has to be processed by a parser. The value has to be in the form "yyyy-MM-dd'T'HH:mm:ss.SSSZ", e.g. "2007-02-12T14:53:00.000+0100" for Monday, the 12th of February 2007 at 14:53:00 CET.

...create and edit non-graphical Supporting Types?

For some supporting types, like a Pool’s Participant or a Message Flow’s Message there are some actions that can be accessed though that element’s context menu (in the BPMN group). For managing other non-graphical elements the EMF tree editor has to be used. Save and close the diagram editor and open the model file with the EMF tree editor. Make your changes to the non-graphical elements, save and change back to the diagram editor. Note that synchronization of the two editors is not supported yet, so you have to close and reopen the diagram editor each time you make changes in the tree editor, so be careful to save your changes in the diagram editor before you change to the tree editor, or you will instantly overwrite the changes done in the tree editor.

A.4 Exporting Business Process Diagrams

Choose the Export feature from the File menu and select the export wizard of choice in the dialog. Select the model files to export – not the diagram files! – and specify a path where to create the target model (see figure A.8). For each BPMN model file a folder with the Business Process Diagram’s name will be created, holding the generated files together with a log file.

![Figure A.8: Model selection page of the BPMN to JIAC Export Wizard.](image)

Note that the diagram has to be in a certain form so it can be successfully transformed. The diagram has to be structured, meaning that must not be blocks or loops with multiple entry or exit points. For each splitting Gateway there has to be a joining Gateway and vice versa. Also note that not each single feature can be taken into account in the transformation yet. For those elements that can not yet be mapped a no-operation element will be created in the target model, such as an empty Activity in BPEL or a `logwarn` in JIAC. Be sure to substitute these elements with a proper implementation.
Bibliography


[38] Stephen A. White. Using BPMN to model a BPEL process. IBM Corp.