



BiZZdesign

&



mendix

NO CODE, JUST GLORY.

MDE in practice: mapping BiZZdesigner to Mendix

MAPPING BIZZDESIGNER TO MENDIX

Master's Thesis

I.R.O. Eeftink

Business Information Technology

University of Twente

14 August 2009

Graduation committee:

Dr. M.E. Iacob

Dr. A. Wombacher

Dr. Ir. H. Bakker

TITLE SHEET

Title:	Mapping BiZZdesigner to Mendix.
Author	Ingmar Eeftink
Student number	0020885
Date	14 August 2009
University	University Twente, Enschede
Company	BiZZdesign B.V. Colosseum 21 7521 PV Enschede www.bizzdesign.com
Graduation Committee	Dr. M.E. Iacob (first supervisor) <i>School of Management and Governance</i> Dr. A. Wombacher (second supervisor) <i>Faculty of Electrical Engineering, Mathematics and Computer Science</i> Dr. Ir. H. Bakker (external supervisor) <i>BiZZdesign B.V.</i>

MANAGEMENT SUMMARY

Motivation

In order to stay competitive, companies are always searching for opportunities to improve their business processes. Techniques regarding business process improvement have developed over time, going from Business Process (Re-)Engineering to Business Process Management (BPM). Current business processes usually involve supporting systems in the form of software, for which requirements are usually created from those models. These BPM models are created by the business side of companies, usually not aimed at software development. Research on Model Driven Engineering (MDE) and Model Driven Architecture (MDA) shows that high level (business) models can be transformed into lower level models, either by hand or automatic. The transition from model to model makes sure that there is always proper communication between the business and the software “worlds”, as well as eliminating shelf-ware models that are forgotten after their creation. Since both sides understand the things they are modelling, less misunderstandings can occur, which leads to reduction of costs and development time to support an agile way of business process development.

This thesis focuses on a specific mapping of a BPM modelling language (BiZZdesigner) to an executable MDE modelling language (Mendix).

Approach

The following research approach is used to address the issue of creating a mapping from the BiZZdesigner language to Mendix: first a literature study is performed to get a clear view on methods for analyzing modelling languages when attempting a mapping. The selected methods are the BWW-ontological analysis and the pattern-based control flow analysis. These analyses are then applied to the case of the BiZZdesigner and Mendix modelling languages, including overviews of matches and mismatches. These results are then used to create guidelines for modelling in BiZZdesigner, if one would want to map their model to Mendix microflows. This includes an example process to explain usage of the guidelines. Lastly, a chapter with a summary and discussion of the results concludes this research.

Results

The results of the BWW and control flow analyses show that there are quite some gaps between the two modelling languages. The BWW analysis’ most striking result was that unlike BiZZdesigner, Mendix lacks the representation for real-life things, which also influences its capacity for system representation. Besides that, a lack of support for system decomposition and differences in lawful transformations with regard to parallelism was found. An overview of capability matches and issues is provided as basis for a mapping from BiZZdesigner to Mendix in section 3.3. The control flow pattern analysis was used to find differences in expressive power of the two languages. This analysis covered

forty-three commonly used behaviour patterns, of which BiZZdesigner matched eighteen and Mendix a mere six. The most striking problem was the lack of support for parallelism in Mendix, which meant quite a few patterns were not supported. A table with matching patterns was provided for the mapping of behaviour in section 4.8, going from the BiZZdesigner language to Mendix.

After analyzing all the matches and problems, the knowledge of these gaps and possible workarounds was turned into guidelines for modelling in BiZZdesigner. The thirteen guidelines are as follows:

1. And-splits and And-joins should be used sparsely. If possible, the modeller should determine the order of otherwise parallel modelled activities.
2. Do not use replicated actions.
3. Use endtriggers in BiZZdesigner whenever a process is supposed to end, avoid implicit ends.
4. Only use one trigger in BiZZdesigner models. Split up the model if there is a need for more than one.
5. Do not use the disable relation.
6. Use normal relations rather than repeated relations for clarity.
7. Do not model real life things, only model items with an appropriate datatype.
8. For clarity, add a separate action in BiZZdesigner that performs the CRUD operation if an action performs it.
9. Only use the repeated action when in a loop. If it is needed to perform the same action a number of times, create a sequence.
10. Only use blocks to separate activities if the content can be seen as a separate process.
11. Do not use endtriggers within blocks, unless the main process should continue.
12. Model system tasks rather than human tasks.
13. Use detailed tasks, but also use layering to keep the top-level model on the business level.

All in all, the mapping from BiZZdesigner to Mendix microflows is less trivial than was expected at the start of this research.

Recommendations and future research

The results of this research lead to the following recommendations and future research directions:

- Research other tools for model execution that may have a better amount of matches in terms of ontology and control flow patterns.
- Research options on how to enforce or automate the presented guidelines.
- Perform more test cases to validate and extend the guidelines.
- Find out how to keep the link between BiZZdesigner originating models and extended Mendix models.
- Research QVT and round-trip engineering.
- Research information mining from Mendix execution model to provide input for BiZZdesigner.
- Perform more research on XPDL and the current implementation of the export to Mendix.

PREFACE

This thesis is the result of the final assignment for the Business Information Technology master at the University of Twente. Much like my time at the university, it has been a project with a few bumps in the road, including a change of plan along the way.

As with every research project, it is not carried out alone and requires feedback and stimulation. For this reason, I would like to thank my supervisors Harm Bakker of BiZZdesign, Maria Iacob and Andreas Wombacher of the University of Twente. I would also like to thank Roald Kruit for his feedback on the Mendix part of this thesis, and everyone at BiZZdesign for the friendly working atmosphere which felt welcoming from the start.

During this project, but also over the years I have spent at the University, there have always been people there for me, whom I would like to thank here for their nearly unconditional support and love: my parents Dick and Willy, my brother Mitchell and my girlfriend Jacqueline. Without them, I would have never come this far.

I would like to finish with an interesting thought on science and education in general:

"The more you know, the more you realise how much you don't know -- the less you know, the more you think you know." - David T. Freeman

August 2009,

Ingmar Eeftink

TABLE OF CONTENTS

1	Introduction	13
1.1	PROJECT BACKGROUND	13
1.1.1	BiZZdesign	14
1.1.2	Mendix	14
1.2	RESEARCH OBJECTIVES AND APPROACH	14
1.2.1	Problem investigation	15
1.2.2	Research questions	16
1.2.3	Research Approach	16
2	Related research	18
2.1	ONTOLOGY	18
2.1.1	Reference ontologies	18
2.1.2	The BWV-model explained	19
2.2	WORKFLOW PATTERNS	21
2.2.1	History	22
2.2.2	Control flow pattern overview	22
2.3	BiZZDESIGNER	23
2.3.1	Domains and model representation capabilities	23
2.3.2	Example BiZZdesigner process	29
2.4	MENDIX BUSINESS MODELLER	30
2.4.1	Mendix domains and representational capabilities	31
2.4.2	Microflows	31
2.4.3	Meta Model	36
2.4.4	Example Mendix model	37
2.5	MODEL DRIVEN ENGINEERING	39
2.5.1	The roots of MDE: MDA	39
2.5.2	Model Transformation	40
2.5.3	BiZZdesigner and Mendix transformations	41
2.6	DISCUSSION	42
3	Applying the BWV-model	43
3.1	BWV MODEL APPLIED TO BiZZDESIGNER AND MENDIX	43
3.1.1	Things and their properties	43
3.1.2	States	44
3.1.3	Events and transformations	45
3.1.4	Systems	46
3.2	OBSERVED DOMAIN CAPABILITY MISMATCHES	48
3.2.1	Construct deficit	49

3.2.2	Construct redundancy	50
3.2.3	Construct overload	51
3.2.4	Construct excess	52
3.3	CONCLUSIONS OF THE BWW ANALYSIS	52
3.3.1	Found matches	54
3.3.2	Construct deficit issues	55
3.3.3	Construct redundancy issues	55
3.3.4	Construct overload issues	56
3.3.5	Construct excess issues	56
4	Behaviour: Control Flow Patterns	57
4.1	BASIC PATTERNS	57
4.1.1	Pattern 1 – Sequence	58
4.1.2	Pattern 2 – Parallel split	58
4.1.3	Pattern 3 – Synchronization	59
4.1.4	Pattern 4 – Exclusive choice	59
4.1.5	Pattern 5 – Simple merge	60
4.2	ADVANCED BRANCHING AND SYNCHRONIZATION PATTERNS	61
4.2.1	Pattern 6 – Multi-choice	63
4.2.2	Pattern 7 – Synchronizing merge	64
4.2.3	Pattern 8 – Multi-merge	65
4.2.4	Pattern 9 – Structured discriminator	65
4.2.5	Pattern 28 – Blocking discriminator	66
4.2.6	Pattern 29 – Cancelling discriminator	66
4.2.7	Pattern 30 – Structured partial join	66
4.2.8	Pattern 31 – Blocking partial join	67
4.2.9	Pattern 32 – Cancelling partial join	67
4.2.10	Pattern 33 – Generalized AND-join	67
4.2.11	Pattern 37 – Local synchronizing merge	68
4.2.12	Pattern 38 – General synchronizing merge	68
4.2.13	Pattern 41 – Thread merge	68
4.2.14	Pattern 42 – Thread split	68
4.3	STRUCTURAL PATTERNS	69
4.3.1	Pattern 10 – Arbitrary cycles	70
4.3.2	Pattern 11 – Implicit termination	71
4.3.3	Pattern 21 – Structured loop	71
4.3.4	Pattern 22 – Recursion	72
4.3.5	Pattern 43 – Explicit Termination	74
4.4	MULTIPLE INSTANCE PATTERNS	74
4.4.1	Pattern 12 – Multiple instances without synchronization	76
4.4.2	Pattern 13 – Multiple instances with a priori design-time knowledge	76
4.4.3	Pattern 14 – Multiple instances with a priori run-time knowledge	77
4.4.4	Pattern 15 - Multiple instances without a priori run-time knowledge	77

4.4.5	Pattern 34 – Static partial join for multiple instances	78
4.4.6	Pattern 35 – Cancelling partial join for multiple instances	78
4.4.7	Pattern 36 – Dynamic partial join for multiple instances	78
4.5	STATE BASED PATTERNS	79
4.5.1	Pattern 16 – Deferred choice	80
4.5.2	Pattern 17 – Interleaved parallel routing	80
4.5.3	Pattern 18 – Milestone	81
4.5.4	Pattern 39 – Critical section	81
4.5.5	Pattern 40 – Interleaved routing	81
4.6	CANCELLATION AND FORCE COMPLETION PATTERNS	82
4.6.1	Pattern 19 – Cancel activity	83
4.6.2	Pattern 20 – Cancel case	83
4.6.3	Pattern 25 – Cancel region	84
4.6.4	Pattern 26 – Cancel multiple instance activity	84
4.6.5	Pattern 27 – Complete multiple instance activity	84
4.7	TRIGGER PATTERNS	84
4.7.1	Pattern 23 – Transient trigger	85
4.7.2	Pattern 24 – Persistent trigger	85
4.8	OVERVIEW	86
4.9	CONCLUSION	90
5	Guidelines for modelling	92
5.1	BWW AND CONTROL FLOW ISSUES	92
5.1.1	Issue 1: parallelism and synchronization	92
5.1.2	Issue 2: replicated actions	94
5.1.3	Issue 3: start and endtriggers	95
5.1.4	Issue 4: differences in relations	95
5.1.5	Issue 5: items and item operations	96
5.1.6	Issue 6: repeated action	96
5.1.7	Issue 7: blocks	96
5.2	GRANULARITY AND MODELLING LEVEL	97
5.2.1	BiZZdesigner modelling level	97
5.2.2	Mendix modelling level	97
5.3	LOSS OF INFORMATION	98
5.4	EXAMPLE MAPPING	99
5.4.1	Registration	100
5.4.2	Acceptance	103
5.4.3	Examination	105
5.4.4	Payment	106
6	Conclusion and Discussion	108
6.1	SUMMARY AND CONCLUSION	108
6.2	RELEVANCE	109

Table of Contents

xi

6.3	LIMITATIONS	109
6.4	FUTURE RESEARCH AND RECOMMENDATIONS	110
6.5	CURRENT IMPLEMENTATION AND RECOMMENDATIONS	110

ACRONYMS

BPM	Business Process Modelling
BPMN	Business Process Modelling Notation
BWW	Bunge Wand Weber
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MDSD	Model Driven Software Development
OMG	Object Management Group
WfMC	Workflow Management Coalition
XML	eXtensible Markup Language
XPDL	XML Process Definition Language

1 INTRODUCTION

In order to stay competitive, companies are always searching for opportunities to improve their business processes. Techniques regarding business process improvement have developed over time, going from Business Process (Re-)Engineering to Business Process Management (BPM). Current business processes usually involve supporting systems in the form of software, for which requirements are usually created from those models. However, the BPM models are created at the business side of companies, aimed at communication and overview rather than software development, which can lead to gaps when developing supporting software systems. Besides that, business demands for software tend to change a lot due to the need for business agility, changing the processes and consequently, the requirements for software that supports the business process. Current globalization also leads to a need for more cross-organizational business processes, which in turn has effect on the need for integration and compatibility on the supporting software.

While BPM support tools matured at the business side, research on software development brought up the concept of Model Driven Engineering (MDE) and Model Driven Architecture (MDA). MDE uses models to lead in development of software, where high level models are transformed into lower level models, either automatically or (partially) by hand. This makes sure that there is always overview and proper communication possible between business process designers and software developers. It also provides for more rapid development of software as it is possible to automate a part of the development process if the models are specified correctly. Besides that, the models always provide a common language for both “worlds” in a MDE project, providing the ability to discuss intended and implemented functionality on the same terms. This leads to less need for communication besides the models and to less misinterpretations as both can understand the models properly. The MDE approach also leads to more flexibility in terms of integration possibilities and maintainability of software, because the models start out platform independent.

This thesis focuses on a specific mapping of a BPM modelling language to an executable MDE modelling language, in order to realize the previously mentioned benefits about creation of supporting software.

1.1 Project background

Mapping the BPM modelling language of BiZZdesigner to an executable modelling language, Mendix microflows, are central in this thesis. If the models can be mapped, use of this mapping should allow for better business/IT alignment and more rapid development of software to support business processes. To gain more understanding of this specific situation, both companies, the tools and languages that are involved will be introduced in this section.

1.1.1 BiZZdesign

BiZZdesign is a spin-off company from the “Telematica Instituut” based on the testbed project, which intended to deliver a design approach and modelling environment for business processes. The project spanned from 1996 to 2001 and delivered structured methods, tools and training for business process engineering. The project proved to be successful, after which BiZZdesign decided to invest in launching these project deliverables on to the market.

BiZZdesign currently consists out of three business units, being:

- Tools, which develops and sells software tools and methods.
- Consultancy, which supports clients when modelling and implementing.
- Academy, which mostly organizes training for tools and seminars.

Currently BiZZdesign's most important tools are: Architect, BiZZdesigner and RiskManager [BIZ08]. For this project, BiZZdesigner is the only tool that will be discussed and used. BiZZdesigner is a tool for (business) process modelling, used for designing, documenting and communicating processes, organisation, information, working instructions and related documentation. In relation to the introduction, it is fairly obvious that BiZZdesigner represents the business process side.

1.1.2 Mendix

Mendix is a spin-off company from the Technical University of Delft and the Erasmus University in Rotterdam, founded in 2005. Mendix released its first commercial product in 2007, and is currently expanding its sales and delivery operations. In 2007 it also opened an office in the United States.

Mendix' product is the Mendix Business Modeller, based on academic work on software generation using model driven application development, going from process models to runtime applications [MEN08]. Mendix does not generate code but instead it interprets models, hence their slogan “no code, just glory”. Relating this product to the previous section, Mendix represents the software development side.

1.2 Research objectives and approach

Following Wieringa's research and design methodology [WIE08], a description and classification of the problem will be given. For this document, the first step of the engineering cycle is provided in section 1.2.1, being the problem investigation in order to create a problem statement. After the problem statement is formulated, this will be broken down into sub-problems, resulting into the research questions posed in 1.2.2.

1.2.1 Problem investigation

In order to come to a proper problem statement, we use the first step of the engineering cycle to generate insight into what the problem really is. This step includes identification of stakeholders, goals, problematic phenomena and criteria for this research project. Then, conclusions can be made for the final problem statement.

Stakeholders

Obvious stakeholders are BiZZdesign and Mendix, as these parties are involved in this project. Other stakeholders are the (potential) customers of BiZZdesign, which have the (latent) wish to map their business process models towards a modelling language closer to an implementation language.

Goals

The goal of the BiZZdesign/Mendix project is to provide their customers and consultants with an easy to use business process modelling tool that is able to create working software from a business process model. To make sure that they reach this goal, BiZZdesign has an interest in knowing the following: *“what problems will arise from mapping BiZZdesigner business processes to the Mendix microflow language?”* For them, the desired outcome of this project is an overview of these problems and a set of guidelines regarding the model mapping to Mendix for business process engineers.

Problematic Phenomena

The problematic phenomena here is that creating a mapping (as in chapter 7 of [KWB03]) from BiZZdesigner to another language like Mendix microflows will probably have specific problems, depending on the capability, control flow and paradigm mismatches of the modelling languages [RM06]. These mismatches most likely have implications for their current modelling techniques, such as requiring additional information to be given or to model the same thing using different concepts. The problems and implications for modelling are to be investigated and documented.

Criteria

The aim is to have as little restrictions as possible on BiZZdesigner modelling when mapping the model to Mendix microflows. The ideal situation would of course be the automatic generation of the Mendix model, given a business process model from BiZZdesigner. However, considering the fact that the modelling style and granularity is on a different level, this is unlikely. Guidelines will need to be created to provide for the information microflows need from BiZZdesigner, as well as possible limitations to the use of certain concepts in BiZZdesigner which Mendix may not be able to handle. These guidelines should be easy to read and usable by BiZZdesigner process engineers.

Concluding: the problem statement

Concluding from the previous sub-paragraphs, the problem statement of this research will be: *“How to transform the BiZZdesigner language into the Mendix microflow modelling language?”*

This problem can be considered a practical problem, more specifically an invention problem as it calls for the creation of a design for future actions.

1.2.2 Research questions

In order to solve the research problem mentioned in the previous section, there is a need to divide the problem into smaller parts. The main issues will be finding out what the BiZZdesigner modelling language is able to do, and if Mendix microflows can match these capabilities. The differences here will lead to a number of restrictions and guidelines that a model mapping to Mendix microflows brings for BiZZdesigner modelling.

This results in the following research questions, which will be further explained in the research approach:

- What are the semantic differences between the two languages?
- What is the difference in expressive power of the two languages?
- How to define a mapping between the two languages in order to support software development?
- How to methodologically support a practical mapping between the two languages?

The sub-questions match the chapters of this thesis as follows: chapter 2 creates the theoretical basis for the analyses that need to be performed to answer the first two questions, as well as providing information about MDE and model transformation in general. Chapter 3 covers the semantic differences in BiZZdesigner and Mendix, followed by chapter 4 which addresses their expressive power. Chapters 3 and 4 lead to a number of possible mapping issues that will be described in those chapters. The methodological support for the mapping, including guidelines, recommendations and an example, is given in chapter 5.

1.2.3 Research Approach

An overview of this research's approach can be found in Figure 1. This section will textually describe the research approach accordingly.

To start with, we need to find out how to properly compare two process languages. Literature shows that the commonly used methods are ontological and control flow pattern analysis. More specifically, the BWV-ontological model [BUN79, WW90] and the control flow patterns of van der Aalst and Russel et al. [AHKB03, RHAM06] will be used. Other methods for analyzing languages are less mature, with little literature support. This will be shown in section 2, along with a detailed description of both analyses. Also, literature about MDE, MDA and model transformations will be studied, in order to be able to create modelling guidelines for a transformation.

To perform the BWV and control flow analyses, information about the BiZZdesigner AMBER [EER99] and Mendix microflow languages will need to be reviewed. This will be done using an

overview of their relevant constructs, as well as an example process for both languages. Where applicable, previous research on model im- and export will be noted.

When all relevant theory is clear, both the BWW and control flow analyses will be performed on BiZZdesigner and Mendix microflows. The results of these analyses will be compared, which will lead to a list of mismatches between the modelling languages. In turn, these mismatches will lead to a number of BiZZdesigner modelling guidelines to properly enable a model mapping to Mendix microflows.

Finally, an example process will be used to illustrate a mapping and its possible issues when going from a BiZZdesigner business process model to a Mendix microflow.

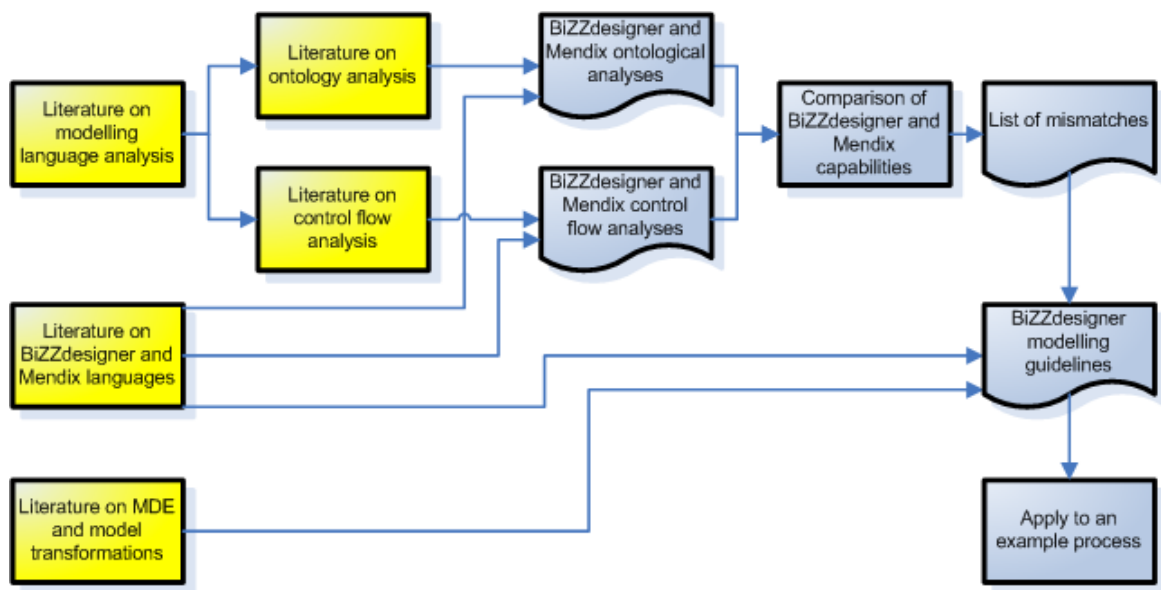


Figure 1: Research Model

2 RELATED RESEARCH

This section will provide an overview of literature that is related to this research. It starts by providing research related to the evaluation of modelling languages, which will be the basis for the analysis of gaps between the BiZZdesigner and Mendix languages. This research will describe ontological and workflow analysis methods. Then, both languages will be described including examples of practical use. In section 2.5 literature about Model Driven Engineering (MDE) is discussed, which is the theoretical basis for using models as a means for transforming a model from one language into another. A brief discussion of the involved literature and its use will end this section.

2.1 Ontology

Ontology is a concept originally taken from the field of philosophy concerned with the study of being or existence [GRU08]. In this philosophical sense, ontology is a *“theory of the nature of existence”*, a definition that dates back to the days of Aristotle. It concerns determining what categories of being are fundamental and asks if, and in what way, items in those categories can be said to “be”. This term was adapted by the Artificial Intelligence scientific community in the 1980’s, followed by a more general adoption by a more general computer science definition. In a very widely cited paper, Gruber [GRU93] defines ontology as *“an explicit specification of a conceptualization”*, which meant *“the objects, concepts, and other entities that are presumed to exist in some area of interest and the relationships that hold among them”*.

When applying this definition to the case at hand, it is clear that an ontology can be used to study the way business processes (the area of interest) are viewed and modelled (the objects, concepts and relations) in both BiZZdesigner and Mendix. Specification of this takes place in the form of definition of representational vocabulary, providing meaning for this vocabulary and constraints for coherent use [GRU08]. In other words, to see if two modelling languages can represent the same concepts, both will need to be compared to a reference ontology. The next section will cover the choice for a reference ontology.

2.1.1 Reference ontologies

A very well-known ontology for information systems is that of Wand and Weber [WW90], who adapted the philosophical ontology of Bunge [BUN77], [BUN79]. Wand and Weber proposed a number of definitions of fundamental information systems concepts, to allow for systematic analysis of modelled information systems. The ontology provided a basis for information system modelling, formalizing concepts using a mathematical notation. Refining this, Wand and Weber came up with a set of models to evaluate modelling techniques by using models for ontological expressiveness (also known as representation model [GR00]), state-tracking and good-decomposition. In this thesis the latter two models are not used, therefore references to the BWV-ontology or BWV-model are henceforth to be seen as a reference to the ontological expressiveness part of the BWV models if not specified differently. This representation model defines a set of constructs that, at this time, are

thought to be necessary and sufficient to describe the structure and behaviour of the real world [GRI05].

Other ontologies, such as the General Ontological Language (GOL) described in for example Guizzardi et al. [GPS05] or the Chisholm ontology in Davies et al. [DGMR05], can also be useful for gaining insights into modelling grammars. However, both of these play only a minor role in the area of ontological analyses [GE07], whereas the BWV-ontology has many examples of analyses being carried out over nearly twenty years. To name some examples, the BWV-ontology has been used to evaluate modelling languages such as the Architecture of Integrated Information Systems (ARIS) [GR00], Business Process Modelling Notation (BPMN) [RIRG05], [RM06], Business Process Execution Language (BPEL) [RM06], Event-driven Process Chains (EPC) [RRIG06] and UML [OH02]. A comparison of various languages in time, from Petri nets to BPMN, can be found in work of Rosemann et al. [RRIG06]. The language evaluation studies not only cover the comparison to the BWV-ontology, but also include examples of comparisons between languages. This broad acceptance and generous literature support makes BWV an excellent choice to use for the case at hand.

That is not to say that the BWV-model is without criticism. For example, a discussion sparked by Wyssusek [WYS06] claimed that the philosophical basis from Bunge of the BWV-model is not feasible and proved a lack of self-reflection in conceptual modelling in general. However, Wand and Weber manage to set aside most of this criticism in [WW06], but also point out that they welcome competing ontologies which could provide a better basis for conceptual modelling. Other criticism comes from Gehlert and Esswein [GE07], arguing that, among other problems, even after nearly twenty years the BWV-model is not interpreted in the same manner by scientists. However, given the broad literature support and relatively immature alternatives, the choice was made to use a BWV analysis in this thesis.

2.1.2 The BWV-model explained

As noted in the previous section, the BWV-model is the best known method to objectively compare, evaluate and determine when to use modelling grammars [GRE07]. The analysis shows the representational capabilities of a modelling grammar in comparison to the reference ontology, providing for a method to check whether a language is able to give a complete and clear description of the domain being modelled. This section will cover the BWV-model in detail, the analysis itself will be carried out in chapter 3 of this document.

The BWV-model contains four categories of constructs [RWR06], being:

- Things, including properties and types of those things;
- States assumed by things;
- Events and transformations that occur on those things;
- Systems structured around those things.

For each construct, there are four possible mismatches which reduce the quality of a language when compared to the BWW ontology. These four are construct deficit, redundancy, overload and excess. Implications of those mismatches on language quality are shown in Figure 2, and will now be described in detail.

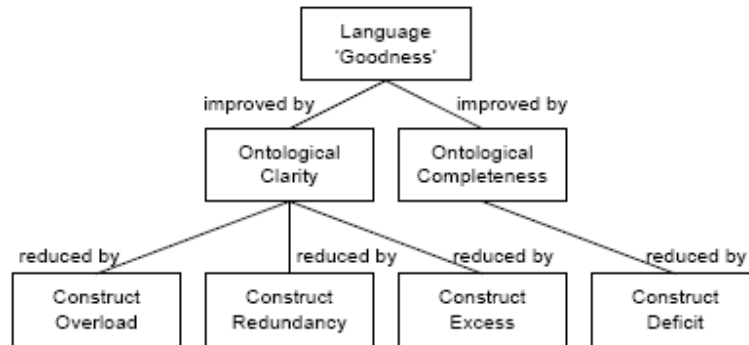


Figure 2: BWW implications for quality of a language [MRI07]

For an overview of these mismatches in relation to the BWW-ontology, see Figure 3.

1. **Construct deficit** means there is a concept in the ontology to which no matching concept was found in the modelling grammar. Consequence of this deficit may be that a modeller is unable to represent certain concepts, and can especially present a problem if a certain modelling grammar supports the concept, but the one it is to be mapped to does not.
2. **Construct redundancy** implies multiple concepts in the modelling grammar representing the same ontological construct. If truly redundant, multiple concepts for the same BWW construct is a weakness in the modelling grammar. However, it is not uncommon to have multiple constructs if they allow one to be more precise (for example subtypes).
3. **Construct overload** means that there are multiple ontology concepts mapped to a modelling grammar concept. In this case it could become unclear how to use a specific modelling construct to realize the ontological concept.
4. **Construct excess** happens when there are concepts in the modelling grammar that cannot be mapped to any BWW concept. When this is observed, these constructs are regarded as excessive, often used to represent something in the problem domain.

These relationships can lead to four measures of quality, being the degrees of completeness, redundancy, overload and excess. Even though these measures are interesting in general, for this research it is most important that BiZZdesigner constructs which match to the BWW-model also have a Mendix equivalent for that BWW concept.

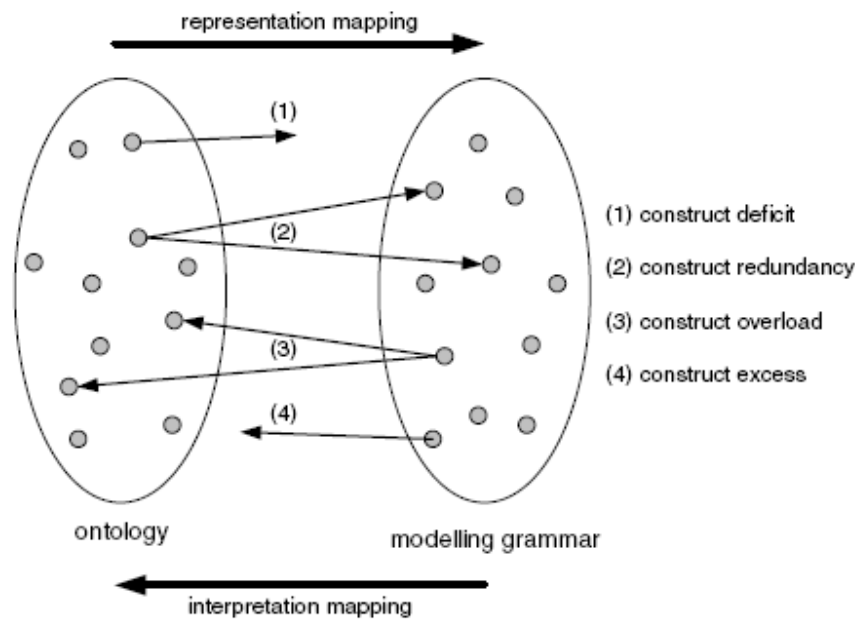


Figure 3: Mismatches from and to the BWW ontology

2.2 Workflow patterns

Besides the ontological analysis, literature suggests combining this with a pattern-based control flow analysis to get a complete picture of the capabilities of a modelling language. Examples of this can be found in an analysis of BPMN [RWR06] by Recker et al. and a comparison of BPMN and BPEL by Recker and Mendling [RM06]. These articles make use of both an ontological analysis and a control flow patterns analysis as a framework for identifying mismatches between process modelling languages. This is supported by literature from van der Aalst [AAL07], who suggests the use of workflow patterns to find out which concepts are desired in an output language when mapping models from one modelling language to another. Use of patterns enables one to determine which functionalities are present in the source language, showing the patterns that should be available in the output language. Pattern based comparison of two languages will expose possible behavioural mismatches when attempting to map models from, in this case, BiZZdesigner to Mendix microflows.

2.2.1 History

The workflow patterns originate from a bottom-up analysis of workflow management software, which was the first to provide an academically supported evaluation for workflow management systems. At first, van der Aalst et al. [AHKB03] propose a set of twenty control flow patterns, which was later extended with another twenty-three in work of Russel et al. [RHAM06]. Along with the development of these workflow patterns, the workflow pattern initiative website [WFP08] was created, currently showing an overview of the full analyses of seventeen languages. Besides the patterns for control flow, three other pattern-based initiatives were developed by the workflow pattern initiative authors. These are data patterns [RHEA04a], resources patterns [RHEA04b] and patterns for exception handling [RHA06]. For this research only the control flow patterns will be utilized, as these will bring insight into the expressive power of the modelling languages involved [RM06].

2.2.2 Control flow pattern overview

The control flow patterns fall into eight separate categories, being basic control flow, advanced branching and synchronization, multiple instance, state-based, cancellation and forced completion patterns, iteration, termination and trigger patterns. An overview of these categories and their corresponding patterns can be found in Table 1.

Workflow Patterns	
<i>Basic Control flow</i>	<i>State based patterns</i>
1. Sequence	16. Deferred Choice
2. Parallel split	17. Interleaved Parallel Routing
3. Synchronization	18. Milestone
4. Exclusive choice	39. Critical Section
5. Simple merge	40. Interleaved Routing
<i>Advanced branching and synchronization</i>	<i>Cancellation and forced completion patterns</i>
6. Multi-choice	19. Cancel Task
7. Structured synchronizing merge	20. Cancel Case
8. Multi-merge	25. Cancel Region
9. Structured discriminator	26. Cancel Multiple Instance Activity
28. Blocking discriminator	27. Complete Multiple Instance Activity
29. Cancelling discriminator	
30. Structured partial join	<i>Iteration patterns</i>
31. Blocking partial join	10. Arbitrary Cycles
32. Cancelling partial join	21. Structured Loop
33. Generalized AND-join	22. Recursion
37. Local synchronizing merge	
38. General synchronizing merge	<i>Termination patterns</i>
41. Thread merge	11. Implicit Termination

42. Thread split	43. Explicit Termination
<i>Multiple instance (MI) patterns</i>	<i>Trigger patterns</i>
12. MI without Synchronization	23. Transient Trigger
13. MI with a Priori Design-Time Knowledge	24. Persistent Trigger
14. MI with a Priori Run-Time Knowledge	
15. MI without a Priori Run-Time Knowledge	
34. Static Partial Join for MI	
35. Cancelling Partial Join for MI	
36. Dynamic Partial Join for MI	

Table 1: Control flow patterns by category

When performing the control flow analysis in chapter 4, a brief description of every pattern will be given. Precise definitions of all pattern behaviours can be found in [RHAM06] and [WFP08].

2.3 BiZZdesigner







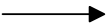
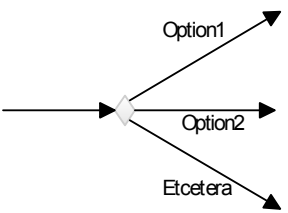
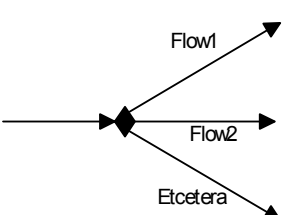
BiZZdesigner is a tool developed during a project of the Dutch *telematica instituut*, originally known as Testbed. The Testbed project ran until 2001, when BiZZdesign was founded as a spin-off of this project. The project goal was to use transparent models and structured methods to achieve systematic and controllable changes in business processes [BIZ03]. The resulting software tool was named BiZZdesigner and kept being developed over the years, up to version 8.7.0 which will be used in this thesis.

2.3.1 Domains and model representation capabilities

The BiZZdesigner tool offers three views, being behavioural, structural (actors) and information (items and data types) [JSL07]. Of these view, behaviour is the most interesting as this is the modelling domain that will be mapped onto Mendix microflows. In this section, the behaviour domain will be described in detail to get an overview of concepts and graphical representation thereof. That is not to say that the domains are not connected, for example the actors are used within activities and item relations can be specified. The relevant connections of these views to behaviour will also be described in this section.

Behavioural view

Behaviour in BiZZdesigner is modelled using actions, which are connected with each other by means of causality relations. Special types of actions are the trigger and endtrigger, which reflect the start and the end of a process. Causality relations can have splits, joins and conditions, all of which will be described in detail in Table 2. Every element has a basic profile which contains information about its name, number, type and which actors are assigned to it. Additional profiles (for example to determine process costs or timing), attributes and scripts can be added by the process engineer.

Element name	Representation	Explanation
Behaviour block		A block containing a set of elements representing a delimited process within the total business process. Behaviour blocks are mostly used to improve readability.
Action	  	<p>Actions are the basic activities in your business process diagram, denoting information in its properties about who/what should be executing the action, how much time it requires, etcetera.</p> <p>Actions can be specified as: standard, repeated or replicated. These three representations are shown on the left in that order. Repetition is caused by a repeated relation, the replication is user defined and specifiable in number.</p>
Trigger		A trigger is an action signalling the start of a process, there can be multiple triggers within a process model.
Endtrigger		The Endtrigger marks the end of a process. A model can have zero to multiple Endtriggers and they can refer to other Triggers in order to start a new process.
Enable relation		The enable relation denoted the order in which activities are performed. Enable relations can be used in various ways, including splits and conditions.
OR-split		An XOR-split which picks a path to follow based on the conditions which are provided by the user. If multiple paths fulfil the conditions, one will be picked randomly.
AND-split		A parallel split which allows for execution of multiple paths. If required, the user can attach conditions to each outgoing enable relation, meaning that not necessarily every path is followed.

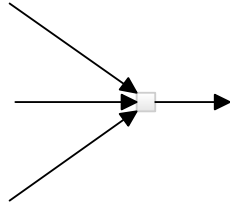
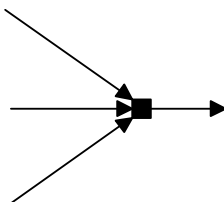
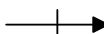
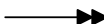

OR-join		Joins multiple flows into one, whenever either of the incoming arches reaches the OR-join, the process after the join is enabled (and other incoming signals are ignored).
AND-join		Joins multiple flows (coming from an AND-split) into one. Every incoming enable relation needs to be activated in order for the next activity to get activated by the AND-join.
Disable relation		The counterpart to the enable relation, which ensures that an action can no longer be executed.
Repeated relation		An enable relation which refers back to a part of the process which has already been executed, allowing for multiple executions of the same process area.

Table 2: BiZZdesigner Behaviour domain

Other relevant constructs to the behaviour domain are the items and item relations and specifying CRUD-operations on those items in the behaviour domain. To illustrate, an example BiZZdesigner process is given in section 2.3.2.

Structural view: actors

Actors perform zero or more behaviour elements, and can be of various types such as a function, role, system, person, etcetera. Actors can be single or replicated, specified in number, where 0 means an infinite amount of that actor. Actors interact with each other using interaction points, which can make use of Item operations. Actors can contain other actors, often used to show actors within organizational units.

Element name	Representation	Explanation
Actor		The basic actor element, including the basic profile with information like name, type and number.




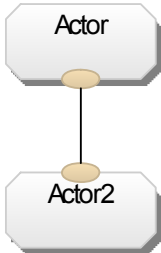
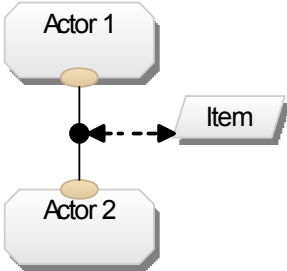
Replicated Actor		A replicated actor specifies the number of actors that is available of that type. If the number 0 is entered, it is shown as *, meaning infinite.
Actor with role or function assigned		An actor element with a role or function profile added to it, which allows for the addition of specific attributes of that profile.
Interaction point	 	Interaction points are used in combination with a interaction point relation (the line in between) to indicate a relation between actors.
Item relation		Items can be utilized using itemrelations, showing which items are used or affected at that interaction (and in that way, every Item operation is available)

Table 3: BiZZdesigner Actor domain

Information view: item relations

Items are data that get used or manipulated within the behaviour domain. Items have a name and can be of a specific data type. The data type is either a standard parameter type like Boolean and String, or of one of the types which are user-specified in the data types domain. Operations on Items are fairly straightforward, basic CRUD operations are supported as shown in Table 4.


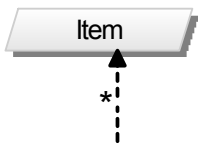
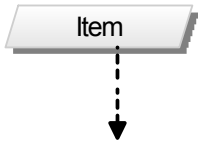
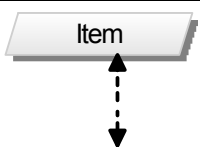
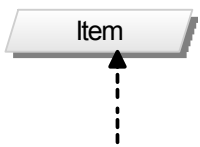
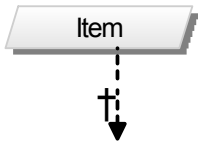

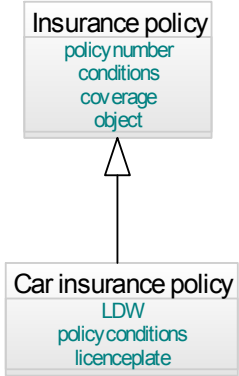
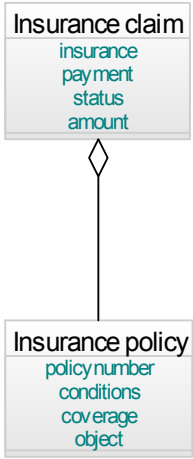
Element name	Representation
Item	
Create item	
Read item	
Read/Modify item	
Update item	
Delete item	

Table 4: BiZZdesigner Item domain

Information view: data types

Data types represent the way data is stored, which can be referred to by Items. A data type has a name and attributes, which can be specified in type. The attributes can be single, a list or a collection. These attributes can be of various standard types, such as Boolean, integer, currency, etcetera, or be of another already defined data type in the diagram.

Element name	Graphical representation	Explanation
Data type		A data type which has a number of attributes and relations. Attributes can have a standard type, or a data type which is already defined. The possible relations are presented in the rest of this table.
Generalization relation		This type of relation is used for inheritance, where a data type refers to another data type as a generalization. On the picture, the car insurance policy is a specialization of an insurance policy.
Aggregation relation		The aggregation relation refers to an attribute within a data type which has an attribute of the type of another data type. For example: the “insurance” attribute in the Insurance claim is an Insurance policy.

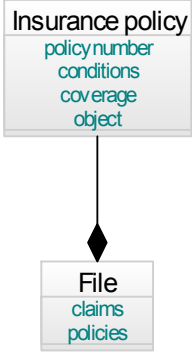
Composition relation		A composition relation means that there is a 0 or 1 to many relation between two data types. For example: a file exists out of polices and claims, the “policies” attribute is a set of insurance policies.
----------------------	---	---

Table 5: BiZZdesigner Data type domain

Aside from the specification of datatypes, there are a number of standard datatypes available in BiZZdesigner. These are: Boolean, currency, date, integer, link, objectref, real, rft, string, time, xml.

To show how all these views and their constructs are used, an example process will be given in the next section.

2.3.2 Example BiZZdesigner process

This example is about an imaginary insurance company called PRO-FIT, often used by BiZZdesign. The complete process model (also discussed in section 5.4) for this case consists out of four main process blocks (see Figure 4), of which the “Registration” will be used in this section to illustrate a typical BiZZdesigner model.



Figure 4: PRO-FIT process blocks

The registration process block is shown in Figure 5 and will be described in detail now. The process starts with submission of a claim (outside the block), which gets received and marked by an employee. Then, relevant customer information is retrieved from a database (the customer file item). A claim is created with relevant damage information extracted from the received claim. Then, a check is done whether the customer is known already or not. Unknown customers get their claim rejected (after which the process implicitly ends), existing customers continue in the process. The process continues by parallel creation of a damage file and registration of claim data, followed by a

completeness check. If the files are not complete, more data is requested and rechecked until complete. After completion, the next process blocks are followed until the claim is paid for in the end.

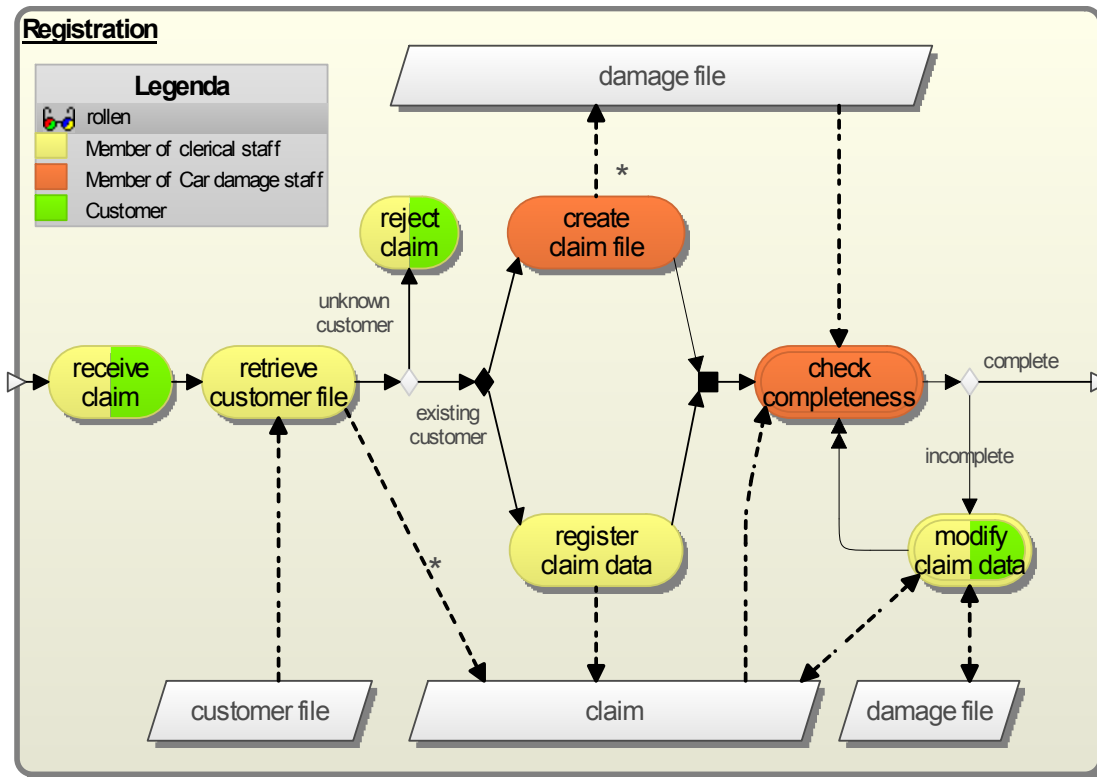


Figure 5: PRO-FIT Registration process block

As can be observed from the various colours, actors are connected to actions. For example, the claim file is created by a member of the car damage staff, whereas the claim data is registered by a member of the clerical staff. The registration block also shows the interaction with items by actions, like retrieval of a customer file. Note that the damage file is represented twice in the model, but is essentially referring to one item. It is only for readability reasons that it is shown twice.

2.4 Mendix Business Modeller

Mendix has created software development platform called the Mendix Business Modeller, which allows for a model driven approach to software development. Their approach is to support both the technical and business side while developing applications, allowing for meaningful communication

between the technical and business side of the company. However, unlike BiZZdesigner the Mendix outset is the software development side, rather than business process modelling.

2.4.1 Mendix domains and representational capabilities

The Mendix Business Modeller has a set of domains in which it operates, these are:

- Micro flows, specifying behaviour of the process flows.
- Meta model, which is basically a data model with classes, properties and their interconnections.
- Forms, to specify things like input, buttons, look, etcetera of forms to show users.
- Resources, allowing for specification of business rules and system rules. These rules are basically limited microflows based on choices, java actions, data sets and enumerations.
- Security, where one can control the rights of actors (and groups of actors) and their CRUD-rights to objects, instances, forms, micro flows and datasets.

In this section the microflow and metamodel domains will be described in detail, as these are the domains where behaviour and data is specified. Since the goal is to find problems when mapping from BiZZdesigner models to Mendix microflows, the form-domain is fairly uninteresting for behaviour specification as well as the security options.



The images and descriptions of the objects in Mendix are based on version 2.3 of the Mendix Business Modeller. At various points, the Mendix wiki [MXDN08] was used to complete the description of some of the elements. The notation used is very similar to the BPMN standard notation.

2.4.2 Microflows

Microflows are used to specify behaviour in Mendix, the types of constructs that can be used in a microflow are: events, (looped) activities, gateways, connecting objects and artefacts.

Events

The most common events are circles in either green or red, which represent the start and end of a micro flow. The start event can only occur once per micro flow, the end event can occur multiple times.

Representation	Name	Explanation
	Start event	This is the start of a micro flow, there has to be exactly one start event in each micro flow.
	End event	Ends the current micro flow. If the micro flow was called within a different flow, it returns to the micro flow that invoked it. There has to be at least one end event in a micro flow.


	Break event	A break event is used within a foreach loop, and is used to “break” out of the loop, usually after an exclusive choice gateway within the loop.
---	-------------	---

Table 6: Mendix Events

Activities

Activities are blue squares in a microflow, which can have various types. The possible types are action calls, object actions, variable actions and client activities. An overview of the possible activities is given in this section.





Representation	Name	Explanation
Action calls		
	Micro flow	Invokes another micro flow within the current micro flow. If the invoked micro flow requires parameters, these should be provided in the Parameter mapping.
	Action	A Java action call which invokes a custom Java action within the current micro flow.

Table 7: Mendix Action calls

Representation	Name	Explanation
Object actions		
	Create	Creates an object of a chosen type, including a variable name which is to be used in that micro flow.
	Change	Changes an available object according to “change member actions” that need to be provided.




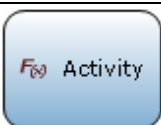
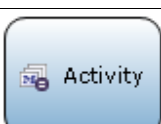
 Activity	Delete	Deletes the chosen available object.
 Activity	Cast	If inheritance was used, a variable can be cast to a derived object type using this activity.
 Activity	Retrieve	Retrieves one or more objects of a particular type, from a database or association. The result can be refined using Xpath constraints and can be sorted using attributes.
 Activity	Aggregate	The aggregate functions sum, avg, count, count, min and max can be used here over attributes of an input variable list.
 Activity	List operation	The list operations union, intersect, subtract, contains and equals can be used here over two input lists.

Table 8: Mendix Object actions


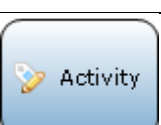
Representation	Name	Explanation
Variable actions		
 Activity	Create	Creates a variable of a chosen type and name, including an initial value. Variables can be created of the following types: Boolean, DateTime, Enumeration, Float/Currency, Integer/Long and String.
 Activity	Change	Changes a chosen variable to a newly desired value, which can be that of a constant, another variable, an attribute, etcetera.

Table 9: Mendix Variable actions



Representation	Name	Explanation
Client actions		
	Show form	Shows a specified form to the user, either replacing the current form (Content) or opening a new one (Popup / Modalpopup, the latter preventing the user to interact with anything on the page except for the popup).
	Send text message	Sends a customizable text message to the user, of the type information, warning or error. The message can be "Blocking" if needed.

Table 10: Mendix Client actions

Lastly, there is a looped activity which iterates over a list of objects (which is the input parameter for the loop), performing the activities which are presented within the "Foreach" square, as shown in Figure 6. A looped activity can contain all elements used in micro flows, except for start and end events.

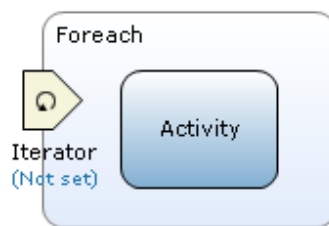



Figure 6: Example of a looped activity

Gateways

Gateways are diamond shaped objects within the micro flow that indicate splits and joins of the process. The three possible gateways are described in Table 11.

Representation	Name	Explanation
	Exclusive split	Basically a XOR-split that splits the flow according to conditions provided in the split. This condition can be based on a variable, an expression or a business rule.



	Exclusive merge	Joins two or more flows back into one (XOR-join) when the unique actions after the split are over.
	Inheritance split	Allows for the use of “super object” properties by sub objects. The split checks the input and chooses the proper outgoing line according to object type. In case it was a sub object, a cast action is required after the split.

Table 11: Mendix Gateways

Connecting objects

There are only two types of connecting objects in Mendix, being the sequence flow and an association. Basically the first indicates the flow between actions, while the second is merely a means to connect comment annotations to other elements.

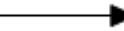

Representation	Name	Explanation
	Sequence flow	The micro flow is executed following the sequence flow arrows, which determine the order in which actions are executed.
	Association	Associations are used to connect annotations to any element in a micro flow except for the start event.

Table 12: Mendix Connecting objects

Artefacts

Artefacts consist out of input data objects and annotations. Their explanation is fairly straightforward and described in the table below.



Representation	Name	Explanation
 (Not set)	Input data object	Specifies the required input variable(s) for the micro flow, of a specified type and name. A micro flow can have zero to many input objects.
	Annotation	Comments that can be attached to elements in a micro flow using an association. Annotations have no impact on the execution and are intended to improve understanding where needed.

Table 13: Mendix Artefacts

2.4.3 Meta Model

The meta model in Mendix is an abstract model (data model), based on the reflection of reality. The meta model consists out of Meta objects, Attributes and Associations. The meta model defines the data structure and can be used to set rights on CRUD (create, read, update, delete) operations.

Meta Object and attributes

A meta object is something that can exist in real life and can have (and usually has) multiple attributes and associations with other meta objects. Examples of these objects are an order, a car, an insurance, a product, etcetera.

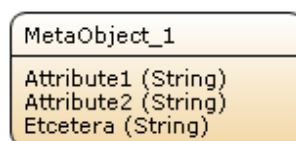


Figure 7: A Meta Object with attributes

An attribute provides information about a meta object. Attributes can store various types of information, these are: AutoNumber, Binary, Boolean, Currency, DateTime, Enum, Float, Hash, String, Integer, Long.

Associations

Associations indicate the relationships between meta objects. An association can be either “basic” or “advanced”.

Basic associations occur when one object just refers to another object, for example a “File” that belongs to a “Customer”. The association occurs if a customer can have multiple files, but a file can only belong to one customer, which basically is the equivalent of a ERD 1-N relation. The relation is recorded in the file, with the cardinality set to 1-0, where the file refers to one customer, and the customer to “0” (zero to many) files.

Advanced associations are either an object reference set or cardinality variations of the basic association. The object reference set means that there is relation in the form of an attribute in a meta object with an object type of another meta object. For example, an Order object could have some attributes, plus a reference set to Orderline objects. The cardinality variations are quite trivial, being a 1-1 object reference (meaning 1 file can have only 1 customer and vice versa) and an M-N or 0-N object reference, meaning a many to many relationship.

Inheritance

As in object oriented programming languages like Java, Mendix supports the concept of inheritance. Inheritance occurs when one meta-object (the sub meta-object) derives the attributes, validation rules

and associations of another meta-object (the super meta-object). A good example of this is a Car insurance policy inheriting from a Insurance policy, as show in Figure 8. Here the Insurance policy holds data like policy number and coverage and the car insurance policy holds information like the licence plate of the car.



Figure 8: Example of Object inheritance

2.4.4 Example Mendix model

To illustrate the presented constructs, this section will provide an example Mendix model. The model is about “Roofmart” which registers customers and their orders. The process is user driven by their portal, existing out of forms, microflows and the Mendix meta-model (basically the data model). The process includes various options, including creation of new customers, new orders and associated order lines specifying the products being ordered. In essence, the complete model consists out of a number of processes which specify the behaviour required for every function. Microflows are advised to be kept small (to one task) [MXT08] and get executed if a button in one of the forms calls it, or if another microflow calls it.

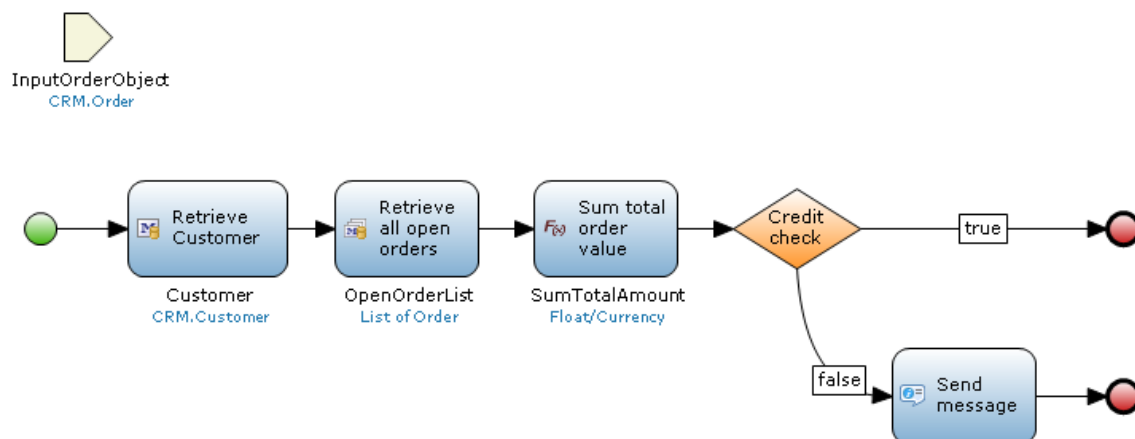


Figure 9: CreditCheck microflow

This example will focus on a small piece of this process, being the credit check which is used when creating a new order. This microflow is shown in Figure 9. First, it finds the customer which is connected to the InputOrderObject and determines its currently open orders in a list. The total value of the currently open orders is determined, followed by a credit check. This credit check is in this case a business rule, shown in Figure 10. The business rule determines whether or not the total amount exceeds that particular customer's credit. If it does not, the rule returns "true", if it does it depends on the customer status what the outcome is (only Gold returns true). If the result is false, a message will be generated stating that the credit check failed and that the account manager should be contacted to see if the customer is eligible for more credit or not.

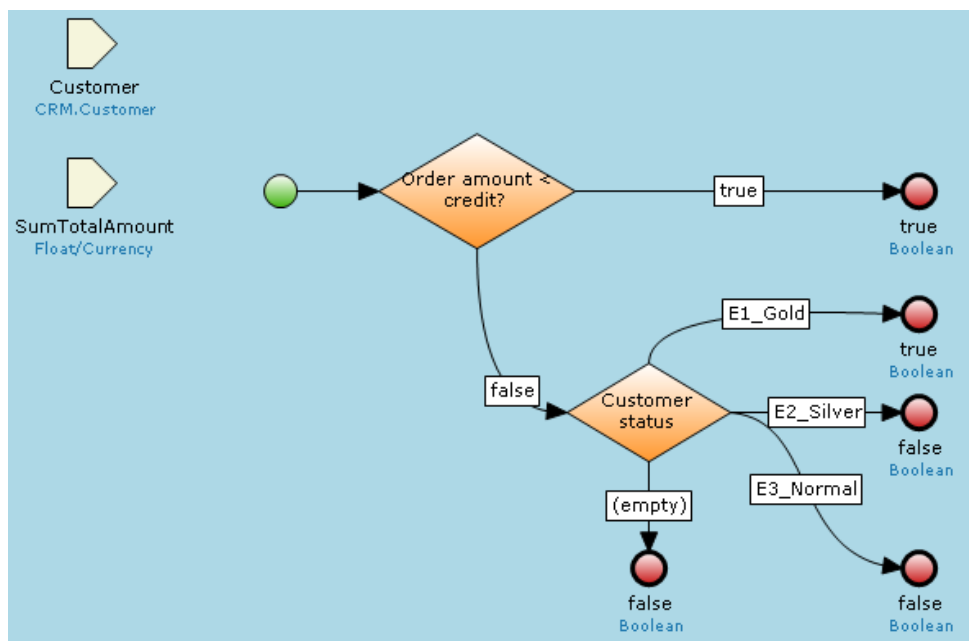


Figure 10: Mendix credit business rule

To illustrate the data context of this example, the Mendix "Meta Model" is shown in Figure 11. In the example, use is made of the Order and Customer objects, along with the OrderLine and CustomerGroup.

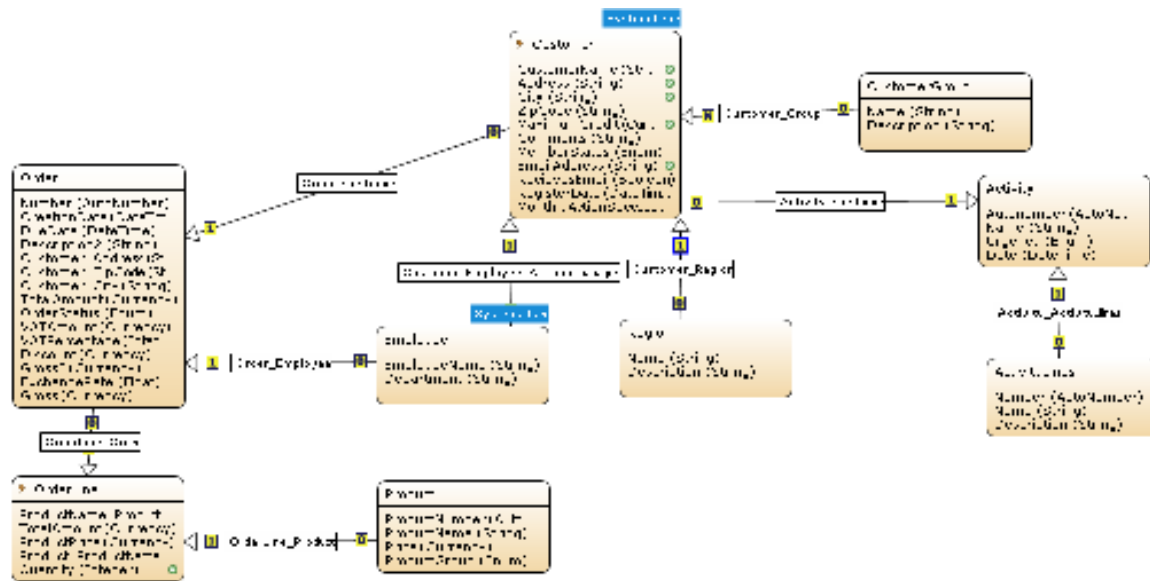


Figure 11: Mendix meta model example

2.5 Model Driven Engineering

Model Driven Engineering (MDE) is a promising approach to handle increasing platform complexity and effective expression of domain concepts [SCH06]. The central thought of the MDE approach is to use models as the basis for software development, in essence using high level business process models as a basis for software design. The term MDE is a fairly recent, originating from work of Kent [KEN02]. Kent provided a framework for MDE, expanding the Model Driven Architecture (MDA) line of thought. Although MDA is more focused on non-business models, it is the origin of MDE and still remains the best known and most widely documented form of MDE.

2.5.1 The roots of MDE: MDA

The initial version of MDA was specified by the Object Management Group (OMG) in 2000 [OMG00] and was revised by various contributors afterwards. MDA originally aimed to separate specification

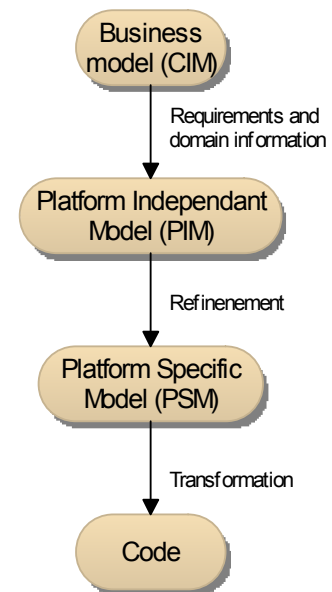


Figure 12: MDA steps

of system functionality from specification of the implementation of that functionality on a specific platform [OMG01]. These two models are called the platform independent model (PIM) and platform specific model (PSM), which have cross model relationships between each other. These relations are usually a refinement that takes place when (preferably automatically) transforming a PIM to a PSM. Above these two levels is the computational independent model (CIM), which is a business model which contains domain information. At the “bottom” of these three models is the executable code, which was preferably automatically generated from the PSM. In most MDA related literature the definition of a model is usually wide enough to incorporate code as being a model as well. An overview of the four modelling steps is given in Figure 12. Note that this is a simplified picture, for example it is possible to also have transformations on the PIM level to refine a model in a PIM to PIM transformation [KWB03].

A second dimension of the level of abstraction is in four layers, called M0, M1, M2 and M3 in OMG's MDA. M0 are the running instances of a system, a process in which for example a specific invoice or order is present. M1 contains models of (software) systems, where for example the concept “invoice” could be specified with associated properties. Basically M1 specifies what instances at the M0 layer should look like. M2 is usually called the meta-model, which specifies the language in which you can model. Examples here would be UML or the BiZZdesigner AMBER language. The M3 level is the meta-meta-model, which is “the model of M2” [KWB03]. This means it is a language to define a modelling language, which includes definition of itself. Within the OMG, the Meta Object Facility (MOF) is the standard M3 language. The differences between these modelling levels is important, as transformations differ for models within the same meta-model and those that are not.

For this thesis, it is imperative to determine what level of modelling will be used when considering the BiZZdesigner to Mendix mapping. The abstraction level of the models is dependant on variation in users and background of tools [BAU08], which can have serious implications when attempting to map models from one language to another. As for the second notion of abstraction, it is clear that BiZZdesigner and Mendix microflows have a different meta-model.

2.5.2 Model Transformation

Model transformations exist in various forms, going from one abstraction level or modelling layer to the another, or a combination of both. This means a model can be transformed at the same level to a different language, it can be refined by going from PIM to PSM, refined on the same level such as a PIM to PIM, etcetera. Any combination of this is possible.

Generally speaking, transformations create a target model from a source model and can be classified in various ways. For starters, transformations can lead to independent or dependent models [GAR03]. The first case means there is no ongoing relationship between the two models, the latter couples the two models by transformation. Transformations can be top-down, if this is the case then changes are always made to the source model and propagated via the transformation. Lastly, transformations can be unidirectional or bidirectional. The unidirectional transformation should allow for changes in the

target model, even if the source model is edited and the transformation performed again. A bidirectional transformation propagates changes in both directions. Bidirectional transformations are a form of round-trip engineering, which prevents models from going out of synch with each other. This form of transformation has issues with loss of information, and the recovery thereof. These losses happen when models are at a different abstraction level, and facilitating bidirectional transformation requires ways to somehow recover this lost data.

Model transformation can be performed manually or automatically. In general, automatic transformations are preferred since those have no possibility for human errors and can always be repeated in the same fashion. However, CIM to PIM model transformations often require human input such as setting parameters for the transformation, and / or manual refinements to the PIM afterwards [FLU06]. MDA provides for automatic model transformations in the form of QVT, meaning Query / Views / Transformation [OMG05]. At this point in time, QVT handles model-to-model transformations on models that comply with a MOF meta-model. Implementations of QVT exist from various sources such as Eclipse M2M [ECL09] and Medini QVT [MED09].

2.5.3 BiZZdesigner and Mendix transformations

Literature on various model transformations has been created during the past decade, however little work on BiZZdesigner and Mendix is available in terms of model transformations.

Currently BiZZdesigner does have various export capabilities to other modelling languages, such as BPEL and staffware. The BPEL export was developed and described by Hoogeveen [HOO06a, HOO06b], which was based mostly on three structured workflow patterns: simple sequence, selection and repetition [WIL77]. To be able to create working BPEL material, the BiZZdesigner process models had to be quite strictly held to various requirements, including specification of a lot of details and a new orchestration actor purely for BPEL. Unfortunately this export was not used very often, probably caused by the need for a large number of changes to existing models, as well as the need for knowledge about BPEL specifics.

More closely related to OMG MDA transformations, research on a BiZZdesigner to OptimalJ model transformation was performed by Jonkers et al [JSLL07]. This was done by first exporting the BiZZdesigner model to XMI using XSLT transformation, followed by QVT model transformations. These QVT transformations not only specified the mapping from BiZZdesigner to OptimalJ's domain model (the PIM), but also PIM to PSM in OptimalJ itself. This research was set out to show the possibilities of adding a model from the BPM world to the MDE world, and therefore lacks formal analyses of both languages involved.

Unfortunately, there is currently no research available about model transformations involving Mendix.

2.6 Discussion

To put all the discussed literature into perspective of the BiZZdesigner/Mendix mapping at hand, a small discussion will show how the theory will be put into use for this research. Most of all, model transformation requires a clear understanding of the abstract syntax and semantics of source and target languages [SK03]. This is why the ontological and control flow analyses will be used, to see how much of their syntax and semantics can be mapped.

The models at hand most likely reside at a different level, since BiZZdesigner is intended for BPM modelling CIM or PIM-level models are to be expected. Mendix is expected to be around PIM or PSM level, as they claim not to create code. This means that not only the meta-model of both languages is different, but also their level of abstraction. Since at the moment the focus is only on going from BiZZdesigner to Mendix without round-trip, the mapping can be classified as a top-down and unidirectional.

3 APPLYING THE BWV-MODEL

In section 2.1 the BWV ontological model was described theoretically, it will now be used to analyze the representational capabilities of BiZZdesigner and Mendix microflows. Not only will the capabilities of both languages be analyzed, a comparison of the capabilities will be made in order to find possible problems for a mapping. This includes construct deficit, overload, redundancy and excesses. These findings will be used in chapter 5 to set up modelling guidelines.

3.1 BWV model applied to BiZZdesigner and Mendix

We will now match the BWV-ontology to the languages of BiZZdesigner and Mendix to see what possible problems there could arise on an ontological level. This paragraph is divided into the four main categories described in section 2.1: things and their properties, states, events and transformations and systems. Each concept within those categories will be briefly described, followed how both languages can or cannot match this concept. After this, an overview of the analysis of both tools in a table will be given.

Descriptions were used from various sources, such as [GR00], [OH02] and [GRE07]. The order in the list of concepts was taken from [GRE07] and falls into the same four categories as [RWR06] as mentioned before.

3.1.1 Things and their properties

Thing

Things are the elementary units in the ontological BWV-model, representing real-world things. In BiZZdesigner things can be seen as the actors and items which can (but not necessarily do) represent real-world things. For example, items can represent an instance of a datatype as well. In Mendix there is no construct for things, only classes of things in their meta-model domain. Unlike BiZZdesigner the actors are only specified by their rights, although they could be the rights of just one person. This however, is not a thing in the BWV sense.

Properties

Real-world things are represented within a model by their properties. In BiZZdesigner, all constructs (including items and actors) have properties such as a name and a standard profile with possible properties to fill out. Only in the Mendix meta-model domain, meta-objects (a class rather than a thing, see below) have specifiable properties.

Class

Classes of things are sets of things with a set of common properties. Classes can be modelled in the datatype domain of BiZZdesigner, creating datatypes with properties to denote a class. Classes can be assigned to items in the process domain. In Mendix, classes are stored in the meta-model domain and are called meta object. The meta objects can be used in the micro-flows and forms.

Kind

A kind is a set of things which can be defined only by their common properties (two or more). This can be mapped to the datatype relations in BiZZdesigner, which allows the user to specify generalization, aggregation and composition relations on classes of things. A kind in BiZZdesigner is also the profile connected to a thing-concept, as this is the representation of the properties of that thing. Mendix has association by reference and reference set relations in their meta-model domain, as well as specification of classes that derive from each other.

3.1.2 States**State**

A state of a thing is defined as *“the vector of values for all property functions of a thing”* [GRE07], meaning that all combinations of possible values of the properties of a thing define its state. There is no construct in BiZZdesigner to model a state of a thing, although it would be possible to check a state using Items to save information about the state. The same goes for Mendix, which lacks a state construct but could make use of a meta-object to save state information manually.

Conceivable state space

The conceivable state space is the set of all states that a thing may assume. Since BiZZdesigner has no state concept, the conceivable state space cannot be viewed or (modelled) either. When using the simulator, the system can be traversed step by step to give insight into the state of the system at every step, but there are no direct ways to model this. The same deficit goes for Mendix, which does not use states.

State law

State laws restrict the values of properties of a thing to a lawful subset based on natural and human laws. Restrictions on properties can be enforced in the BiZZdesigner action domain where laws can be specified in the condition profile of an enabling relation or by using script operations. In the datatype domain, this is possible by specifying cardinalities on relations. It is possible to create restrictions on properties of meta-objects in Mendix, in the form of validation rules. It is also possible to specify cardinalities on the relations between meta-objects.

Lawful state space

Lawful state spaces and events that can conceivably occur in the thing. In BiZZdesigner, it is possible to specify restrictions on properties. The resulting remaining combinations of variables of the things are the lawful state space. This space however, is not directly viewable or possible to model in BiZZdesigner. The situation is analogue for Mendix.

3.1.3 Events and transformations

Events

Events are actions that involve changes on things. Events can be internal or external, these will be described in detail later on. Specification of events in BiZZdesigner is only possible for Item-operations since there are no events specifiable in the behaviour diagram that involves changes on the properties of actors. Mendix does not offer specification of things, but it would seem that the various operations it offers on objects and variables (see Table 8 and Table 9) covers about the same operations as BiZZdesigner on this matter.

Conceivable event space

The conceivable event space is the set of all possible events that can happen within a thing. This set is not directly specifiable in both BiZZdesigner and Mendix.

Transformation

A transformation is the mapping of each state of a thing to another, lawful state of that thing. BiZZdesigner is not state-based, but the transformation of the state by going from one action to the next by enable-relations matches this. The specification of the action in combination with possible transformation laws (see below) in the following relation should match a transformation definition of BWV. Mendix is much like BiZZdesigner here, using action types plus their specification with possible constraints.

Lawful transformation

A lawful transformation is a law which defines what transformations are lawful. These laws are a property of a thing. Transformation laws are governing events, defining the allowed changes of states.

Datatypes have the possibility to restrict values of their properties, limiting the possible states of an Item instantiating that datatype. In BiZZdesigner the AND and OR-splits can also define “laws” which determine what path is chosen (lawful).

Mendix’ meta-objects can be restricted in their values. Mendix’ exclusive split has rules to determine which path is chosen, with a possibility to refer to one or more business rules.

Lawful event space

The set of events which is allowed since there are no laws that prevents them from happening (a subset of the conceivable event space). This set is not specifiable in BiZZdesigner, it is the space which is not restricted by laws that can be specified. This also goes for Mendix.

History

History is comprised out of the states that a thing goes through chronologically ordered in time. This concept is not present in BiZZdesigner, although you can check the traversal through a process by making use of the simulation tool. The simulator also allows for steps back and saving the scenario a

user choose to create while going through the simulation. Mendix does not offer a history concept either. There is some support in the tooling in the form of choosing steps in a DOS-box while in debug-mode, however the language itself has no support for history.

Acts-on / Coupling

Things are said to act-on one another if the existence of a thing affects the history of another thing. Things are coupled if they act on each other.

In BiZZdesigner things can be coupled by using the interaction points and possible item relations within the actor domain. Also, things can be coupled in the action domain if for example an action performed by an actor which affects an Item. Since Mendix does not offer specification of things, this cannot be matched. Class-dependencies are possible to see though.

3.1.4 Systems

System

A system is set of things which, if bi-partitioned in any way, has coupling among things within the two subsets. The BiZZdesigner process model is, if it has incoming (enabling) relations between every action and item except for the trigger(s), a system. The same goes for the actor domain, which should have connecting points and relations between actors and possible item relations. For Mendix a system is not supported, since there are no directly visible connections to actions and its users. The only visible relation is possible user access.

System composition

The things in a system are the system composition. In BiZZdesigner, these are the actors and items, since Mendix is based on classes of things rather than things, this does not match.

System environment

Things which are not in the system, but interact with other things within the system. The environment in BiZZdesigner can be an external actor which can interact with other actors or perform an action on an item via an action which has the actor connected to it. At first glance, users are the system environment in Mendix, these are present in the form of "system.user" which can provide for input when editing classes. However, since the user is not directly affecting a thing, but a class or variable, this is not support by this definition.

System structure

The system structure is the set of couplings that exist among things inside the system and those in the environment of the system.

BiZZdesigner has system structure in three domains. In the action domain the enable/disable relations, including the possible splits and joins and item relations. The actor domain the coupling consist out of the connecting points and relations between actors and item relations where applicable.

Data domain: the various relations possible between datatypes: generalization, aggregation and composition.

The Mendix microflows offer sequence flows like the enable relations in the BiZZdesigner action domain. Mendix also has the same kind of structure in their meta-model domain as in the data domain of BiZZdesigner.

Subsystem

Subsystems are systems which are a subset of another system in terms of composition and structure. Use of a block in the action domain of BiZZdesigner singles out a piece of the entire process and can thus be seen as a subsystem. A microflow in Mendix can call another microflow, which seems comparable to using blocks. However, blocks in BiZZdesigner only hide a view on certain activities and things, whereas Mendix has no overview of an entire system like BiZZdesigner (no “fold out” options). The microflow that is called is truly outside the flow in which you are calling it, making it doubtful if this is truly a subsystem in the BWW sense.

System decomposition

The system decomposition is a set of subsystems where *“every component in the system is either one of the subsystems in the decomposition or is included in the composition of one of the subsystems in the decomposition.”* If the entire process model in BiZZdesigner is made out of blocks containing only one construct, the system is completely decomposed. Mendix is unable to specify system decomposition.

Level structure

A level structure is a partial order over the subsystems in a decomposition to show which subsystems are components of other subsystems or the system itself. The block structure in BiZZdesigner enables level structure in the action domain. The actor domain also allows for this by creation of actors within other actors. Level structure is not possible in Mendix. By calling other microflows in an overview of actions the behaviour could be comparable, but this is not entirely the same.

External event

An external event arises within a thing, system or subsystem by an action in the environment by aforementioned concepts. The state before this type of event must be stable, the state after can be unstable or stable. A trigger in the BiZZdesigner action domain represents an external event, the Mendix equivalent is the start event.

Stable state

A state in which a thing, system or subsystem will remain until acted upon by an external event. There will be a stable state when the endtrigger, or when an action with no outgoing arches is reached in BiZZdesigner. Also, one could create a stable state situation like this by using a trigger and an AND-join in the process flow, enforcing the process to remain there until the trigger is enabled. In Mendix, the only stable state is at the End Event.

Unstable state

A state in which a thing, system or subsystem can be, but will change into another state by an action of transformation in the system (internal event). Basically the BiZZdesigner process is in an unstable state until the endtrigger is reached, or the above mentioned stable states. However, there is no construct to specify which type of state the process will be in. Mendix has a comparable situation, as long as the End Event is not reached, the system is in an unstable state.

Internal event

An event that arises within a thing, subsystem or system which is a lawful transformation. The before-state of this type of event is always unstable and can have a stable or unstable after-state. BiZZdesigner actions can perform internal events. The endtrigger can invoke a trigger of another, or the same, process. Mendix action types can perform internal events as well, using the various action types. This includes invocation of other microflows within a microflow.

Well-defined event

Given an initial state, the next state is deterministically determined after this type of event. Actions (events) without splits after them are well-defined events in BiZZdesigner. This situation is comparable to Mendix, with the difference that the presence of gateways determines whether the event is deterministic or not.

Poorly defined event

Poorly defined events are events which, given prior state, cannot predict the subsequent state. BiZZdesigner can have chances attached to outgoing arches of splits, which means the next state cannot be predicted in such a case. Mendix can create the same kind of behaviour using an expression in an exclusive gateway with a random chance in it.

3.2 Observed domain capability mismatches

Based on the analysis done in the previous section, an overview of the capability mismatches of both BiZZdesigner and Mendix will now be provided. This will be done using the four measures of quality that were given in section 2.1, being construct deficit, redundancy, overload and excess. Doing so should yield insight into what BWV-concepts can be mapped directly by showing a one to one relation to a BWV concept for both BiZZdesigner and Mendix, and to show what concepts are missing in Mendix (deficit), but are available in BiZZdesigner. Furthermore construct redundancy and overload are important, as these make it hard to match a concept to another. For example, if BiZZdesigner has an overloaded a construct matching two BWV concept while Mendix has separate constructs for these two BWV concepts, it will be hard to map this BiZZdesigner construct to Mendix without additional information. Construct excesses also need to be addressed, since the excesses in BiZZdesigner will need to be handled somehow when mapping to Mendix.

3.2.1 Construct deficit

The construct deficits represent BWB concepts that have no match to a construct in the modeller grammars of BiZZdesigner or Mendix.

For BiZZdesigner, the most striking deficit is in the states section. States are not directly specifiable, although it is possible to limit the states of things by means of laws that restrict property values. Since BiZZdesigner is a process tool rather than a state-based tool this is quite logical, but it may cause problems and may require workarounds with actions that check for an artificially created “state” property. Besides the deficits on states, it is also not possible to specify the conceivable or lawful event space, which means that these spaces will have to be thought up by the users themselves. Another deficit is the inability to have a view on the history of a thing, although this can partially be done by using the simulation option within BiZZdesigner.

The most important deficit in Mendix are “things”, this deficit affects the event, acts-on and system constructs as well. These things could be an important issue when transferring the models between the two tools. Mendix is also missing a match to the BWB state concept, leading to the same issues as mentioned for BiZZdesigner in that regard. Lastly, there is no concept in Mendix which is like the BiZZdesigner Block, thus the tool does not allow for subsystems, a level structure and system decomposition.

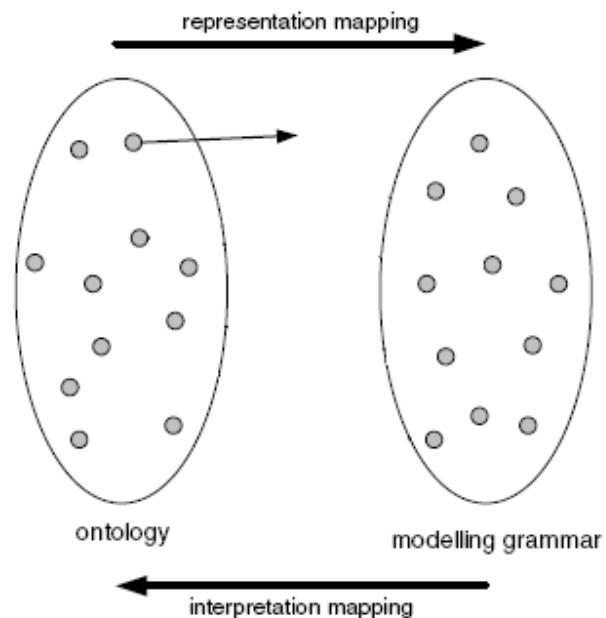


Figure 13: Construct deficit

3.2.2 Construct redundancy

Redundant constructs are a multitude of modelling grammar constructs that map to one BWV concept, basically offering multiple ways to represent the same.

To start with BiZZdesigner, there are two constructs for things, begin Items and Actors. Since the difference between these two concepts seems quite obvious, it is unlikely that a user would experience problems keeping them apart. Items are things that can be created, updated or deleted by actions, while actors are the performers of actions, not being affected by those actions.

A “kind” can also be of two types, but are in different domains (data and actor domains), meaning it is also unlikely these will get mixed up with one another. Events can be specified on two different domains as well, in the action domain it is primarily to show which Items are affected by actions, while in the actor domain the events are used to show what actors perform actions on items. This is redundant in the sense that actions in the action domain have actors coupled to them, meaning that this type of information is already available in the action domain. Much like the events, the “acts-on” concept is also present in the action and actor domain.

A system is specified mostly in the actor domain, where all “things” are present without the actions. However, deriving the system from a connected process model is possible (actor property on every action, and items with relation to those actions), leading to a redundancy. This type of issue continues at the system composition and environment. Lastly, datatype restrictions and rules enforced in splits decide which transformation is lawful, which is a redundancy.

Concluding, the BiZZdesigner redundancy issues seem mostly related to the ability to specify items and actors within both the action and actor domain. Basically an actor schema could be drawn from the action model if properly specified.

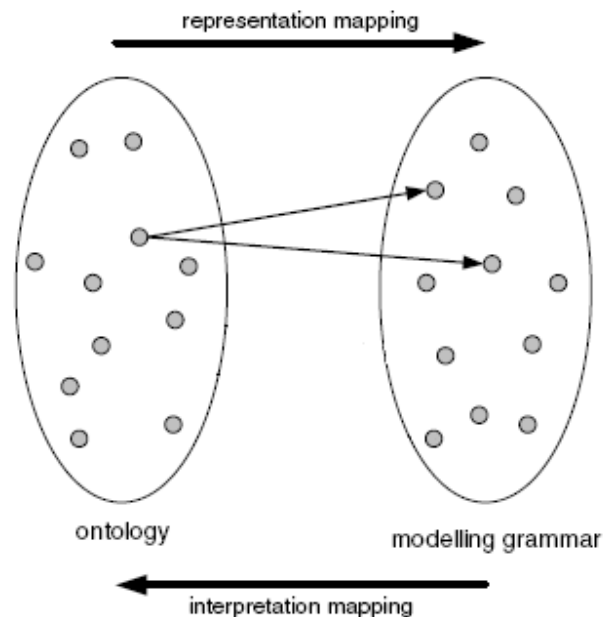


Figure 14: Construct redundancy

There is redundancy in Mendix as well, starting with state laws and lawful transformations. A state law can be either a validation rule or a cardinality connected to meta-object relations. Lawful transformations are specified by the various action types, as well as rules in the exclusive split, potentially connected to or specified in business rules.

Structure is noted in meta-model associations and inheritance. There are also sequence flows interconnecting actions. The “kind” concept also has two ways to specify the same type of association. Lastly, internal events need to be specified and have various representations depending on the type of action.

3.2.3 Construct overload

Construct overload is about the same modelling grammar concept matching to multiple BWV concepts.

Concept that turn up in mapping from BiZZdesigner to multiple BWV-concepts are the Items, Actors and Blocks. The Item and Actor concepts are “overloaded” as they represent the things as well as the system composition. Since by definition the system composition is the things within a system, while the system is a connected process model, this overlap would seem to have little impact.

The Blocks in BiZZdesigner can be used to create subsystems, a system decomposition and to create a level structure. This means that blocks can be used to

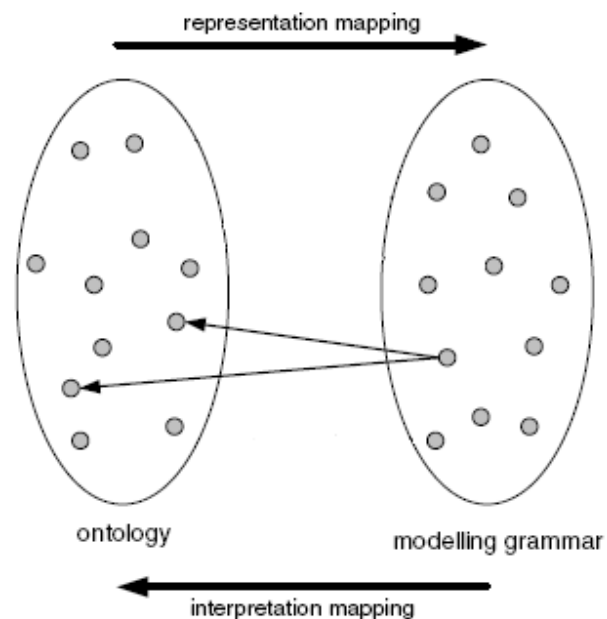


Figure 15: Construct overload

Overload in Mendix was not found.

3.2.4 Construct excess

Constructs excess are the constructs in the modelling grammars which could not be mapped to any BWW concept.

In BiZZdesigner the construct excess is found in the repeated relation, the repeated and replicated action types and comments.

Mendix excess consists out of the looped activity (foreach), the association connecting object, break event and the artefacts, being input data object and annotation.

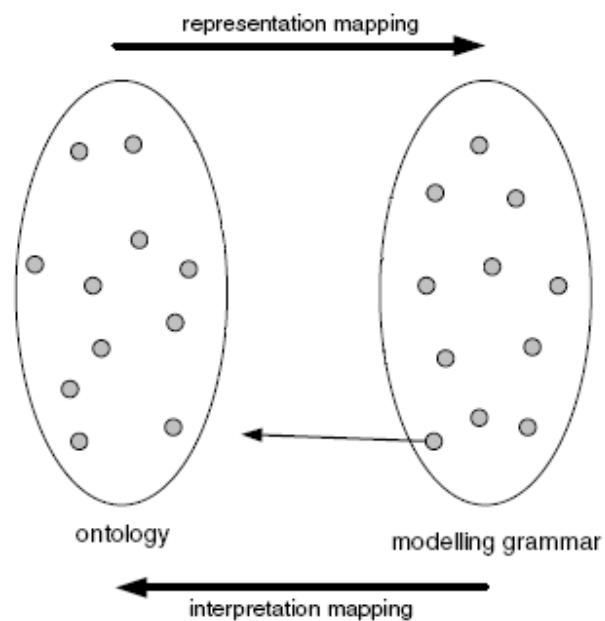


Figure 16: Construct excess

3.3 Conclusions of the BWW analysis

Now that the BWW analysis is complete, the next step will be to describe the implications of the mismatches between BWW, BiZZdesigner and Mendix. First, an overview of supported (+) and not supported (-) BWW constructs will be given in Table 14. In this table a * means there is construct redundancy, which is of interest should models be mapped from one to another. We will then continue with a description of the BWW-matching issues when considering the mapping from BiZZdesigner to Mendix. At the end of this section construct excess of BiZZdesigner will be addressed, this excess should be mapped to Mendix via another route than BWW.

Ontological construct	BiZZdesigner	Mendix
Thing	+*	-
Property	+	+
In general		
In particular		
Hereditary		
Emergent		
Intrinsic		
Non-binding mutual		

Binding mutual Attributes		
Class	+	+
Kind	+	+
State	-	-
Conceivable State space	-	-
State Law	+	+
Lawful state space	-	-
Event	+	-
Conceivable event space	-	-
Transformation	+	+
Lawful transformation	+	+
Lawful event space	-	-
History	-	-
Acts-on	+	-
Coupling: binding mutual property		
System	+	-
System composition	+	-
System environment	+	-
System structure	+	+
Subsystem	+	-
System decomposition	+	-
Level structure	+	-
External event	+	+
Stable state	+	+
Unstable state	+	+
Internal event	+	+
Well-defined event	+	+
Poorly defined event	+	+

Table 14: BWW model as in Green et al. [GRE07] applied to BiZZdesigner and Mendix

3.3.1 Found matches

Using the BWW analysis, there are some direct construct matches that can be found. These can be observed in Table 14 by a “+” for both tools. There are also some matches that can be made to a construct with redundancy or overload, which will also be described in this section.

Classes, properties and kind

The meta model domain in Mendix and the data domain in BiZZdesigner can be mapped nearly one to one. The classes can be mapped to meta objects and the same type of relations (including kind) between the classes are possible.

Transformation

The transformation in both tools are similar, both using an arrow (the enable relation and sequence flow) between actions/activities with the possibility to specify a transformation law.

Event

Even though the events cannot be matched since Mendix cannot operate on “things” like BiZZdesigner, a match could be made if the item that BiZZdesigner is operating on has a datatype. The match can be made to the action type that represents the CRUD operation on the item in BiZZdesigner. Note that the read/modify item operation is just a combination of a read and update operation, which is conform to the retrieve and delete action in Mendix.

Internal event

The BiZZdesigner actions can perform internal events, while the same goes for Mendix action types. Even though BiZZdesigner does not have a specification of action types, actions can always be mapped to the general action in Mendix without a specific type.

External event

The trigger of both languages can be mapped to each other via the external event. A note must be made that there can be multiple triggers in BiZZdesigner while only a single trigger is possible in Mendix.

Well-defined and poorly defined event

In both languages, a well-defined event is an event which has no splits behind it. The only difference is that BiZZdesigner can have two types of splits, while Mendix only has one gateway option.

Poorly defined events are the exact opposite, having a chance to their outgoing split arches. This goes for both languages, again with the difference that BiZZdesigner offers a parallel split and Mendix does not.

Stable and unstable state

Both languages are in an unstable state until they are at the point that no further actions can be performed. For Mendix this is more clear than for BiZZdesigner, as in Mendix this is always the case

for the end event while BiZZdesigner can also have implicit ends without use of an endtrigger. Note that if the BiZZdesigner model has two performing two branches, the endtrigger does not end the process until both branches are finished.

3.3.2 Construct deficit issues

For a mapping to Mendix, construct deficits in BiZZdesigner are no problem. Therefore, only the deficits that Mendix has while BiZZdesigner supports the construct will be discussed in this section. The first problem is “things”, which BiZZdesigner can specify in items and actors. Even though the items can have a datatype, these can also represent real-life things which means that they cannot be represented in Mendix. The same goes for actors, which can represent real-world persons rather than a set of rights. The problems with BWW things are again shown in events, which by definition involve things.

In the system category Mendix has deficit issues as well, which are all related to things and the block-construct in BiZZdesigner. These issues are about representing subsystems, decomposition and level structure.

Problematic concept	Reasoning
Thing	Things can be represented in BiZZdesigner, but not in Mendix. When attempting a mapping, information will likely be lost.
States and state laws	States are not properly mapable to the BWW ontology for both languages and state laws are specified differently (script language / expressions)
Events	Events happen on things and since Mendix cannot represent those, there is a problem here. Some transformations that are lawful in BiZZdesigner cannot be created in Mendix (AND-split/join for example)
System	Mendix has no block-like construct to match the decomposition / system structure and fails to match on “things” again, such as the system structure.

Table 15: Overview of mapping issues after BWW analysis

3.3.3 Construct redundancy issues

As shown in section 3.2, there is quite some redundancy in both modelling languages, including some BiZZdesigner constructs that are not present in Mendix. These issues are found in the things and systems category, which we have shown in the previous section to be missing in Mendix.

However, redundancy issues that are not associated with deficit problems can also be found. These are state laws and lawful transformations, BWW concepts that are redundant in both BiZZdesigner and Mendix. Since they are not redundant in the same fashion, this will likely cause problems when mapping the two languages.

Lastly, the specification of events in Mendix requires more information about the type of action that is to be performed. However, there is also a “blank” action-type which could be used when mapping the construct.

3.3.4 Construct overload issues

Less clearly visible from the table are construct overload issues. Much like at the redundancy section, there are overload issues for BiZZdesigner which are already “covered” by a deficit in Mendix. This includes the block, item and actor constructs.

3.3.5 Construct excess issues

Obviously there is no problem for Mendix to have construct excess when compared to the BWW ontology. However, for the mapping from BiZZdesigner to Mendix the excess of the former needs to be clear, possibly with a matching to the Mendix excess concepts. It is possible that the Mendix construct excess matches the BiZZdesigner excess.

The first excess in BiZZdesigner is found in the repeated relation and action, which basically is a way of showing that the relation or action can be enabled multiple times within the same process instance. Since this kind of behaviour is the same when using a normal enable relation in BiZZdesigner, this does not seem to be a problem. In Mendix, creating similar behaviour is not specifically shown by a same type of relation, it is just represented as a “normal” sequence flow. Secondly, the replicated action is a way to show the start of the same action in parallel. This will obviously be a problem considering the fact that Mendix unfortunately does not offer parallel functionality. Lastly there is excess in the form of comments, which can be matched to the Mendix annotations.

Problem	Reasoning
Repeated relation	The repeated relation has no mapping to a BWW construct, however as there is another way in BiZZdesigner to represent the same behaviour (just using a normal enable)
Repeated action	This is quite similar to the repeated relation, only showing that it is possible that the action has the possibility to be performed multiple times. This has no further implications
Replicated action	The replicated action is performed (a pre-set) number of times, which are started in parallel.
Comments	Basically these are human-readable information, who do not represent a (formal) specification of behaviour.

Table 16: Construct excess issues

4 BEHAVIOUR: CONTROL FLOW PATTERNS

To find out which concepts are desired in an output language when mapping, van der Aalst [AAL07] suggests the use of workflow patterns. This way, it is possible to determine which functionalities are currently in BiZZdesigner and thus preferred as possible in the output language. This will expose possible problems when attempting to map models from BiZZdesigner to Mendix. First, we will determine which patterns are available in the BiZZdesigner AMBER language and in Mendix. This will be done following the order of the workflow pattern initiative [WFP08].

After the pattern analysis, a comparison will be made of the capabilities of BiZZdesigner 8.7.0 versus the 2.3 version of Mendix microflows to find gaps. Solutions or workarounds to these gaps will then be provided where possible.

4.1 Basic patterns

The first class of patterns is the basic patterns, which capture the elementary aspects of process control. Even though an explanation is given for each pattern, the document of Russel et al. [RHAM06] and the Workflow Patterns initiative [WFP08] can be consulted more detailed descriptions.

Since these are the basic concepts, it seems obvious whether they are supported or not. In the case of BiZZdesigner, all five patterns can be found in the synonyms of the pattern names, which means the explanation for support is rather trivial. Table 17 shows the support for all basic patterns.

Pattern		BiZZdesigner	Explanation
1	Sequence	+	Enable relation after a previously completed task
2	Parallel split	+	The AND-split
3	Synchronization	+	The AND-join
4	Exclusive choice	+	The OR-split
5	Simple merge	+	The OR-join

Table 17: Basic patterns in BiZZdesigner

Unfortunately, not all basic patterns are supported by Mendix. The omission of a parallel concept implies the lack of support for patterns 2 and 3, and hurts the support for various other patterns as well.

Pattern		Mendix	Explanation
1	Sequence	+	Activities are started after the completion of a previous activity or start event via a sequence flow.
2	Parallel split	-	Parallelism is not supported in Mendix.
3	Synchronization	-	Since parallelism is not supported in Mendix, this

			construct does not exist.
4	Exclusive choice	+	The Exclusive split gateway.
5	Simple merge	+	The Exclusive join gateway.

Table 18: Basic patterns in Mendix

4.1.1 Pattern 1 – Sequence

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

This pattern is implemented fairly straightforward in BiZZdesigner, by using an enable relation from one activity to another. In this case, activity a enables b (Figure 17).

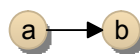


Figure 17: Pattern 1 in BiZZdesigner

The sequence pattern in Mendix is trivial like in BiZZdesigner, being one activity connected to another activity via a sequence flow (Figure 18).

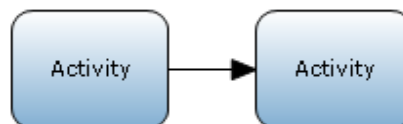


Figure 18: Pattern 1 in Mendix

4.1.2 Pattern 2 – Parallel split

A point in a workflow process where the process' active branch is split into two or more parallel branches, which allows for simultaneous execution of activities (or in any order).

The parallel split is easily realized in BiZZdesigner by using the AND-split construct. In the example of Figure 19, three activities are activated upon completion of activity a. The activities b, c and d are then executed concurrently and can be synchronized afterwards by pattern 3.

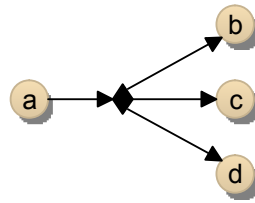


Figure 19: Pattern 2 - Parallel split in BiZZdesigner

As mentioned earlier, patterns 2 and 3 about parallel split and synchronization are not supported by Mendix. The only option to get around this is to create a Java action which performs the required parallel activities.

4.1.3 Pattern 3 – Synchronization

A point in a workflow process that merges multiple active branches (which are usually created earlier in the process by a parallel split) into a single branch, which is only enabled when all incoming branches have been enabled.

Compliant with pattern 2, synchronization is also supported in BiZZdesigner. In Figure 20 the three activities b, c, d are synchronized by use of the AND-join construct. Activity e is executed after all previous activities have completed.

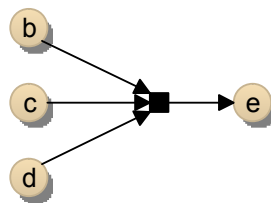


Figure 20: Pattern 3 – Synchronization in BiZZdesigner

Patterns 2 and 3 about parallel split and synchronization are not supported by Mendix. The only option to get around this is to create a Java action which performs the required parallel activities.

4.1.4 Pattern 4 – Exclusive choice

The divergence of a branch in a workflow process into two or more branches, where exactly one of those outgoing branches is enabled after evaluation of the condition provided in the choice.

The exclusive choice is realized by using the OR-split construct in BiZZdesigner. Figure 21 is a simple example of this, where either b, c or d are activated depending on the condition(s) specified at each outgoing enable relation.

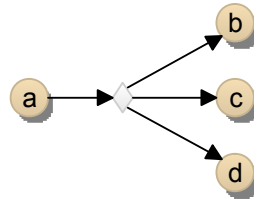


Figure 21: Pattern 4 - Exclusive choice in BiZZdesigner

For Mendix, the exclusive split gateway provides the exact behaviour described, which is following one of the options given based on evaluation of the condition within the split.

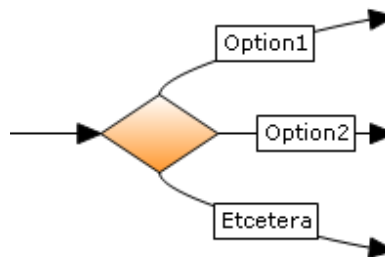


Figure 22: Pattern 4 – Exclusive choice in Mendix

4.1.5 Pattern 5 – Simple merge

A point in a workflow process that merges multiple branches into a single branch, which is enabled when one incoming branch has been enabled.

The simple merge is also available in BiZZdesigner, by use of the counterpart of the OR-split: the OR-join. The example in Figure 23 shows an activity e which is started when either of the three incoming activities are completed. Activity e will not be activated more than once, even if for example both b and c are completed.

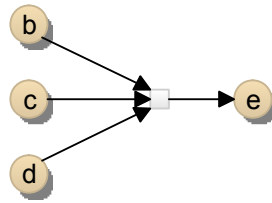


Figure 23: Pattern 5 - Simple merge in BiZZdesigner

The exclusive join is represented by an exclusive join gateway in Mendix, being the counterpart of the split provided earlier.

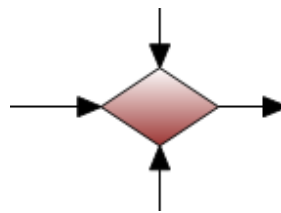


Figure 24: Pattern 5 – Simple merge in Mendix

4.2 Advanced branching and synchronization patterns

This class of patterns focuses on more complex choices and merging of processes. Even though most of these patterns are quite common in practice [WFM08], they are not always properly supported.

This is also the case in BiZZdesigner, which supports only two of the four initial patterns (6 through 9) and none of the more advanced versions.

	Pattern	BiZZdesigner	Explanation
6	Multi-choice	+	Using and AND-split with conditions, as is shown in Figure 25.
7	Structured synchronizing merge	-	Basically you need to check the conditions twice in such a pattern, which means there is no direct support for this pattern.
8	Multi-merge	-	The OR-join only expect to receive 1 input, and only enables the following processes once
9	Structured discriminator	+	AND-split -> OR-join, the OR-join in

			Bizzdesigner puts out 1x enable and ignores the second.
28	Blocking Discriminator	-	Blocking not supported.
29	Cancelling discriminator	-	Not supported, BiZZdesign's OR-join cannot cancel preceding activities once it is activated
30	Structured partial join	-	Even though the behaviour can be specified using AND / OR-joins, there is no direct support for this pattern
31	Blocking partial join	-	This pattern is basically a combination of pattern 28 and 30, which are not supported.
32	Cancelling partial join	-	Pattern 29 and 30 seem to be combined here, not supported.
33	Generalized AND-join	-	BiZZdesigner does not store the amount of enables that were received.
37	Local synchronizing merge	-	There are no means of providing an OR-join with information about when it should enable the following branch.
38	General synchronizing merge	-	Lack of a possibility to check which branches are activated except for a second check of split conditions.
41	Thread merge	+	Outgoing arch after a replicated action
42	Thread split	+	Replicated action

Table 19: Advanced branching and synchronization patterns in BiZZdesigner

Advanced branching and synchronization patterns are not properly supported by Mendix because of the missing parallel construct mentioned at the basics section. Even though the structured discriminator is supported, it will never be used as there will not be parallel activities going on. The more advanced types of branching and synchronism patterns are not supported by Mendix either because of the lack of support for parallelism described previously. A note must be made that a lot of these patterns could be implemented in a Java Action, however this is not direct support.

Pattern	Mendix	Explanation
6 Multi-choice	-	Parallelism is not supported in Mendix, thus activation of more than one path is not an option.
7 Structured synchronizing merge	-	Since parallelism and pattern 6 are not supported, this pattern is neither.
8 Multi-merge	-	The exclusive join construct only expects one input.
9 Structured discriminator	+	Even though parallelism is not supported, the exclusive join works this way.
28 Blocking Discriminator	-	Since parallelism is not supported, neither is this

			pattern.
29	Cancelling discriminator	-	Since parallelism is not supported, neither is this pattern.
30	Structured partial join	-	Since parallelism is not supported, neither is this pattern.
31	Blocking partial join	-	Since parallelism is not supported, neither is this pattern.
32	Cancelling partial join	-	Since parallelism is not supported, neither is this pattern.
33	Generalized AND-join	-	Since parallelism is not supported, neither is this pattern.
37	Local synchronizing merge	-	Since parallelism is not supported, neither is this pattern.
38	General synchronizing merge	-	Since parallelism is not supported, neither is this pattern.
41	Thread merge	-	Since parallelism is not supported, neither is this pattern.
42	Thread split	-	Since parallelism is not supported, neither is this pattern.

Table 20: Advanced branching and synchronism patterns in Mendix

The only supported pattern by Mendix here, pattern 9, cannot be shown since it would require at least two active branches.

4.2.1 Pattern 6 – Multi-choice

Much alike the parallel split, a multi-choice is a point in a workflow process where the process' active branch is split into two or more parallel branches, which allows for simultaneous execution of activities (or in any order). However, the multi-choice allows for specification of conditions like in the exclusive choice pattern, meaning that not necessarily every outgoing branch has to be activated.

The multi-choice pattern is currently supported by BiZZdesigner, by using an AND-split with conditions attached to the outgoing arches (see Figure 25). The OR-split cannot be used, as it will only enable one outgoing arch. The example given in Figure 25 can enable one, two or three arches depending on the value of x . The four different types of input values for x , with their outgoing relations are:

- $x < 6$ → Only enables the arch to c
- $x = 6, 8, 9$ → Enables arches to b and c
- $x = 7$ → Enables all three arches
- $x > 10$ → Only enables b

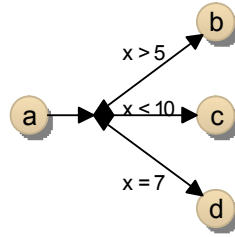


Figure 25: Pattern 6 – Multichoice in BiZZdesigner

4.2.2 Pattern 7 – Synchronizing merge

The synchronizing merge is the counter-part of the multi-choice pattern, merging multiple branches into one. The next branch should be activated when all earlier diverged branches that were active have enabled the synchronizing merge.

There is no direct construct for the synchronizing merge in BiZZdesigner, however using a second check that checks which branches were enabled earlier, or by use of disable relations one could attain this behaviour. In Figure 26, two examples of a workaround are given to realize this pattern. The first checks the condition of the multichoice after activity a again, while the second uses disable relations to disable the path if both

$c1 \ \& \ \neg c2 \rightarrow \langle a, b, d \rangle$
 $\neg c1 \ \& \ c2 \rightarrow \langle a, c, d \rangle$
 $c1 \ \& \ c2 \rightarrow \langle a, b, c, d \rangle \text{ or } \langle a, c, b, d \rangle$

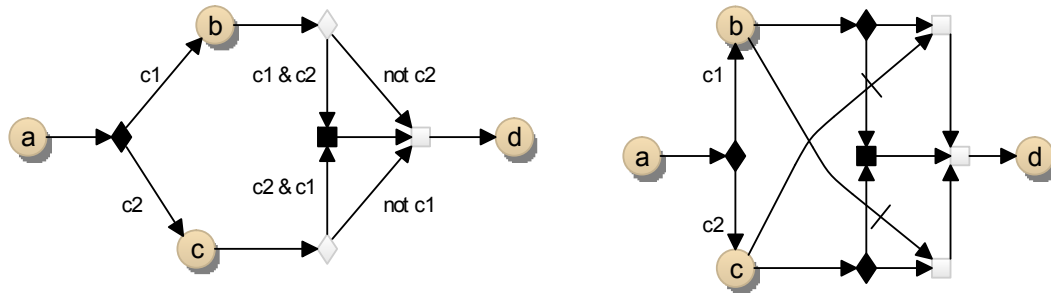


Figure 26: Pattern 7 - Structured synchronizing merge in BiZZdesigner (not supported)

4.2.3 Pattern 8 – Multi-merge

Like the simple merge pattern, a multi-merge is a point in a workflow process that merges multiple branches into a single branch, which is enabled when one incoming branch has been enabled. However in this pattern, each incoming enablement should result in a thread of control being passed to the subsequent branch rather than just once.

This pattern is not supported by BiZZdesigner, as the OR-join only passes on one enable if it receives enables from incoming arches (and ignores the rest). To illustrate (Figure 27) pattern 8 should have the following output if we decide to start with a, d: <Start, a, d, b, c>. If the pattern was supported, it should be <a, d, b, c, d>, including a second execution of activity d.

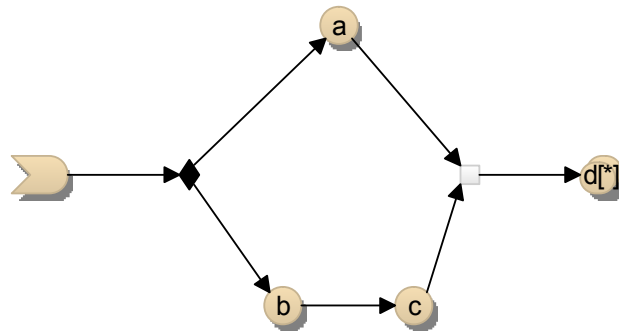


Figure 27: Pattern 8 – Multi-merge in BiZZdesigner (not supported)

4.2.4 Pattern 9 – Structured discriminator

A structured discriminator converts two or more incoming branches into one, passing the thread of control to the following branch when the first incoming branch has been enabled. Subsequent enablements from incoming branches do not pass on a thread of control which was the case in the multi-merge pattern.

The structured discriminator is directly supported in BiZZdesigner in the form of the OR-join. As said, the activity after the structured discriminator should not be activated again after having been activated once. This is shown in Figure 28, where b or d can activate the OR-join first, the second activation being discarded by the join.

<Start, a, b, e, end>
<Start, a, c, d, e, end>

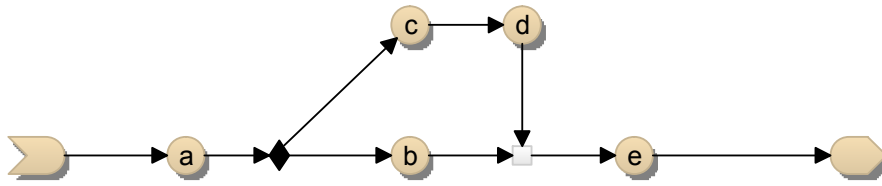


Figure 28: Pattern 9 - Structured discriminator in BiZZdesigner

4.2.5 Pattern 28 – Blocking discriminator

Like the structured discriminator (Pattern 9), two or more incoming branches are converted into one, passing the thread of control to the following branch when the first incoming branch has been enabled. The difference is that subsequent enables of the discriminator are now blocked until all incoming active branches have enabled the discriminator.

It is not possible to prevent creation of a new process instance in BiZZdesigner as is required by this blocking discriminator.

4.2.6 Pattern 29 – Cancelling discriminator

A cancelling discriminator converges two or more branches into one, enabling the following branch whenever one of the branches enables the discriminator. In this pattern, the discriminator cancels all other incoming branches when activated by one.

This pattern is not supported in BiZZdesigner, it is not possible to specify a “cancelling” version of the OR-join. It is also impossible to create disable relations from splits/joins, meaning this behaviour cannot be constructed.

4.2.7 Pattern 30 – Structured partial join

The convergence of two or more branches (say m) into one, which enables the following branch when a fixed number of branches (say n) have been enabled. Here n is a specifiable number and is required to be smaller than m , leading to the term “ n out of m join”.

In Figure 29, there need to be at least two completed actions from the set a, b, c to enable the AND-joints which in turn enables an OR-join, allowing the process to continue. This is a simulation of a structured partial join where $n=2$ and $m=3$. However, since there is no direct construct (no way to specify the number of required incoming enables to an AND-join), the support for this pattern is not

present. One could also imagine how things would look if it would be a 9 out of 10 join, creating loads of unreadable “spaghetti”.

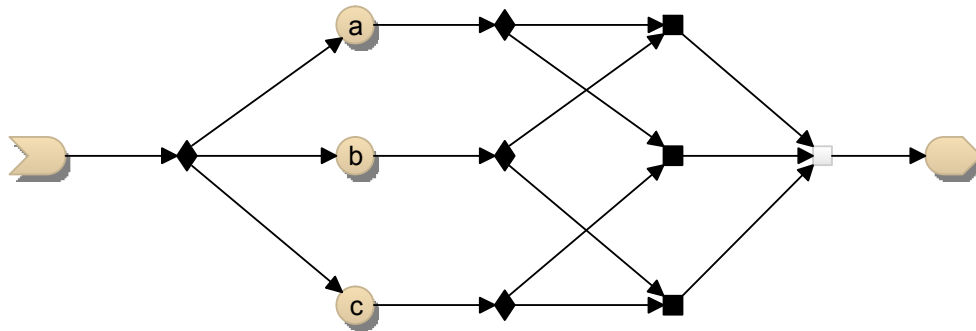


Figure 29: Pattern 30 – Structured partial join emulation in BiZZdesigner (2 out of 3 join)

4.2.8 Pattern 31 – Blocking partial join

Like the blocking discriminator, but enables the following branch whenever n incoming branches have been triggered.

This pattern is a combination of pattern 28 and 30, which are both not supported by BiZZdesigner, leading to the fact that this pattern is not supported either.

4.2.9 Pattern 32 – Cancelling partial join

Basically a cancelling partial join is a variation of the structured partial join, where other incoming branches are disabled as soon as the specified number of branch enables is reached.

This pattern is basically a combination of pattern 29 and 30, which are both not supported by BiZZdesigner, and neither is this one.

4.2.10 Pattern 33 – Generalized AND-join

The generalized AND-join is like the synchronization pattern (Pattern 3 – Synchronization), but adds persistence for the amount of received triggers.

This pattern is not supported by BiZZdesigner, it does not store the amount of enables that were received.

4.2.11 Pattern 37 – Local synchronizing merge

The local synchronizing merge is a variation of the structured synchronizing merge pattern (pattern 7) where the number of incoming branches that require synchronization is determined based on locally available data to the merge construct.

In BiZZdesigner, there are no means of providing an OR-join with information about when it should enable the following branch (it just enables the next arch after one incoming enable). The fact that pattern 7 already was not supported only adds to the conclusion that this pattern is not supported in BiZZdesigner.

4.2.12 Pattern 38 – General synchronizing merge

The general synchronizing merge is a variation of the structured synchronizing merge (pattern 7), which merges multiple branches into one. In this case however, the next branch should be activated when all earlier diverged branches that were active have enabled the synchronizing merge or can never be activated in the future.

Since BiZZdesigner does not offer support for pattern 7 because of lack of a possibility to check which branches are activated except for a second check of split conditions, it is apparent that this pattern is not supported either.

4.2.13 Pattern 41 – Thread merge

The thread merge pattern merges a specified number of execution threads in a single branch of the same process instance into one thread of execution.

BiZZdesigner supports this pattern by use of the enable relation after a replicated action, the arch out is not activated until all instances are done.



Figure 30: Thread split and merge in BiZZdesigner

4.2.14 Pattern 42 – Thread split

Thread splits are the initiation of a nominated number of instances of execution threads in a single branch of the same process instance.

The replicated action can be specified to perform a number of instances of an activity.

4.3 Structural patterns

This type of patterns focuses on the possible restrictions there might be on models with regard to cycles and need for explicitness of termination. More complicated forms (21/22/43) of these patterns are also included in this section.

Pattern		BiZZdesigner	Explanation
10	Arbitrary cycles	+	Cycles can be created using the repeat-relation
11	Implicit termination	+	A process automatically ends when there are no longer any actions to perform, even if there is no explicit endpoint reached.
21	Structured loop	+	It is easily possible to create this behaviour, using the repeated relation with an OR-split which checks if there needs to be another loop pass or not.
22	Recursion	-	It is not possible to refer to a process in an action. A reference to start a new instance can be made using the “Trigger” value in the properties of an Endtrigger, but this only starts a new instance of (possibly the same) process, independent of the parent process.
43	Explicit Termination	-	The end trigger construct does not cancel remaining activities, a process only ends when there are no more enabled actions.

Table 21: Structural patterns in BiZZdesigner

Structural patterns in Mendix are relatively well supported, with the exclusion of implicit termination and recursion.

Pattern		Mendix	Explanation
10	Arbitrary cycles	+	Arbitrary cycles can be constructed using exclusive splits and joins, as shown in Figure 32.
11	Implicit termination	-	A decisive end event is required.
21	Structured loop	+	There is a direct construct for this: the looped activity (Figure 35), which iterates over a specifiable input variable.
22	Recursion	-	A micro flow can invoke itself in an activity within the flow, however it will wait for the invoked process to complete before continuing

			itself. See Figure 37 for an example, where “AddAgainViaMicroFlow” is the name of the flow itself.
43	Explicit Termination	+	A sequence flow leading to and end event ends the process (or takes it back to the micro flow that invoked the current micro flow).

Table 22: Structural patterns in Mendix

4.3.1 Pattern 10 – Arbitrary cycles

Arbitrary cycles is a pattern which allows for cycles with more than one entry or exit point.

Arbitrary cycles ensure the possibility of repetition in a process model, the implementation in BiZZdesigner is the same as presented at [WFM08], as shown in Figure 31. The repeated relation can go either directly into the actions, or via an OR-join. For example, the process could loop back like <start, a,b,c,b,a, etcetera>.

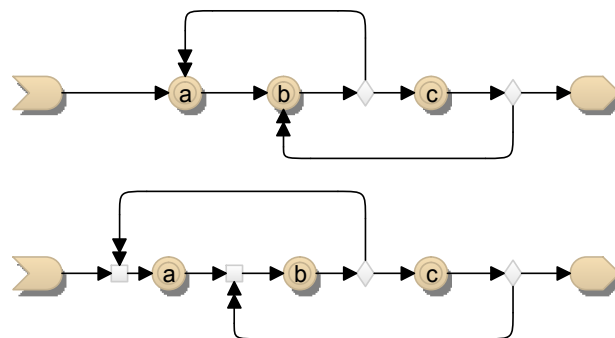


Figure 31: Pattern 10 – Arbitrary cycles in BiZZdesigner

Arbitrary cycles are supported in Mendix by just combining splits and joins, much like in BiZZdesigner. The pattern is shown in Figure 32.

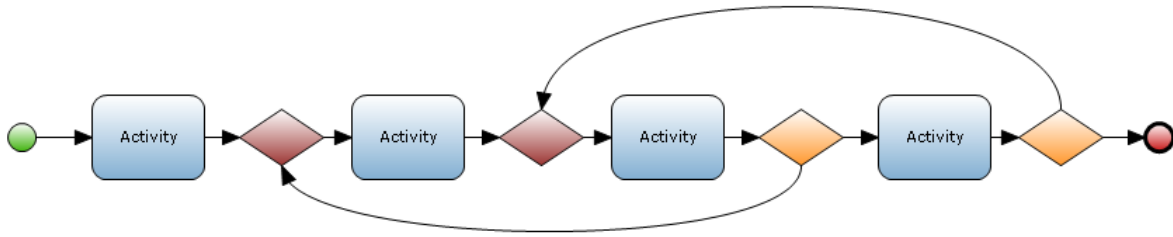


Figure 32: Pattern 10 – Arbitrary cycles in Mendix

4.3.2 Pattern 11 – Implicit termination

Implicit termination means the end of a process (instance) if there are no remaining activities to do at that point in time or at any time in the future.

As shown in Figure 33, pattern 11 is supported. If action b is selected at the OR-split, the process ends after b is completed without ever activating an endtrigger.

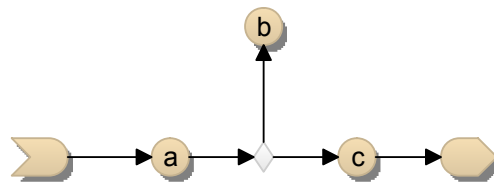


Figure 33: Pattern 11 - Implicit termination when activity b is done

4.3.3 Pattern 21 – Structured loop

A structured loop exhibits the ability to execute a task or sub-process repeatedly. It is a loop with a pre-test or post-test condition, which determines what it should do either at the beginning or at the end of the loop respectively. The loop in this pattern has a single point of entry and exit.

The structured loop is supported by BiZZdesigner, where you can use an OR-split with conditions and a repeated relation to create a structured loop. In an example of this behaviour in Figure 34, action a is done until the condition specified is false, in which case the process will continue to action b.

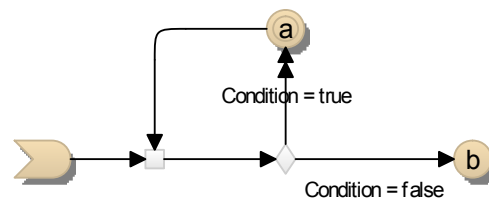


Figure 34: Pattern 21 – structured loop in BiZZdesigner

The looped activity in Mendix iterates over specifiable input, conform pattern 21 (see Figure 35).

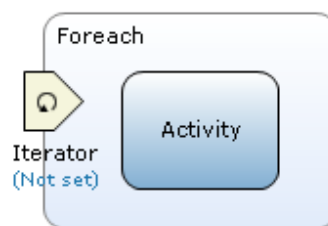


Figure 35: Pattern 21 - structured loop in Mendix

4.3.4 Pattern 22 – Recursion

The ability of a task to invoke itself during execution, or an ancestor.

Even though it is possible to invoke another instance of the same process in an endtrigger, it is not possible to do so in an action, meaning that completion of a process instance would not be reliant on the invoked process instance, as is the case when using recursion.

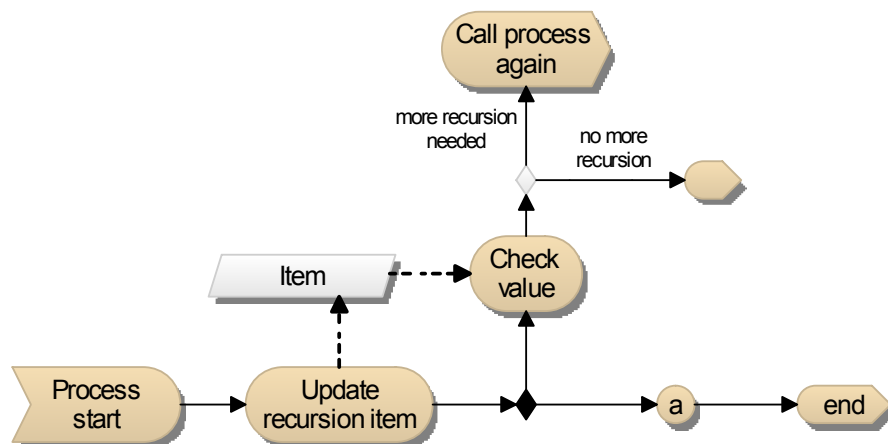


Figure 36: Pattern 22 - Recursion in BiZZdesigner

Recursion cannot be realized by having an activity invoke the micro flow it is in, as the activity will wait for it's completion before it will start the next (Figure 37). A possible workaround would be to create a java class and use this class within a microflow by use of the java action call.

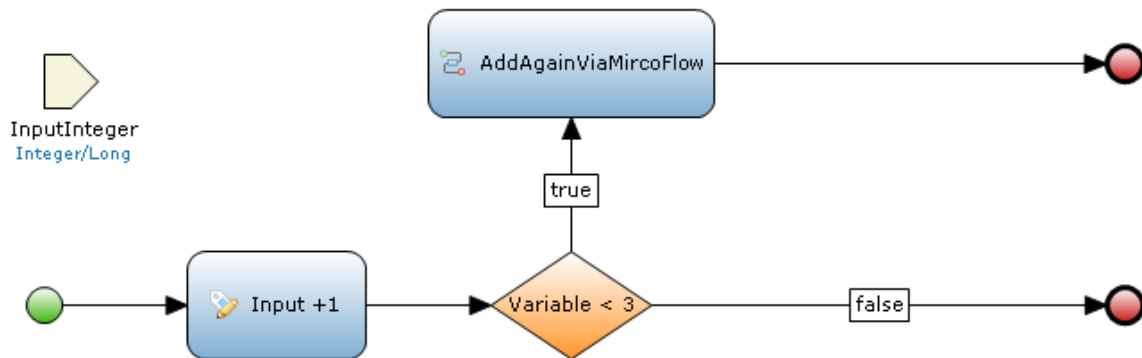


Figure 37: Pattern 22 - Recursion in Mendix (not supported)

4.3.5 Pattern 43 – Explicit Termination

Explicit termination is if a process instance ends when it reaches a certain state, usually in the form of an end note. Any remaining work should be cancelled if this state is reached.

BiZZdesigner uses implicit termination and has no real explicit termination construct. The endtrigger which could be expected to behave according to this pattern does not cancel remaining work.

Explicit termination is modelled using the End event in Mendix, which was already described in section 2.4.2.

4.4 Multiple instance patterns

As the name clearly indicates, these patterns are about multiple threads of execution within process models. BiZZdesigner supports the basic patterns (12 through 15) fairly well, but fails to represent any of the more advanced multiple instance patterns.

Pattern		BiZZdesigner	Explanation
12	Multiple instances without synchronization	+	Using action replication you can create multiple instances (specifiable in number), which are not synchronized using an AND-join afterwards.
13	Multiple instances with <i>a priori</i> Design-Time knowledge	+	Replicated action with number of instances, making copies of the same process and using an AND-split and an AND-join reflects the same

			behaviour, shown in Figure 39.
14	Multiple instances with <i>a priori</i> Run-Time knowledge	-	There is no way to implement this behaviour directly, however using a counter with scripts one could realize it. In Figure 40, action a sets the number of instances in an item, b reads this item and creates the correct number of instances
15	Multiple instances without <i>a priori</i> run-time knowledge	+	An action checks if another instance is required, or if it can continue with the rest of the process. See Figure 41.
34	Static partial join for multiple instances	-	As an extension of pattern 13, if modelled as a replicated action this pattern cannot be done, if modelled as separate actions this could be checked using a construction like the pattern 30 emulation, however there is no direct support for this pattern (and this would get unreadable when considering a 10 to 7 example).
35	Cancelling partial join for multiple instances	-	BiZZdesigner does not support pattern 34 already, meaning this pattern is not supported either.
36	Dynamic partial join for multiple instances	-	Dependant on pattern 14 and much like pattern 34. Since we do not take scripts into account, there is no support for this pattern.

Table 23: Multiple instance patterns in BiZZdesigner

Multiple instance patterns require multiple instances of activities to be active. Mendix does not provide a construct to create multiple instances, or even to emulate them using an AND-split. Therefore, unfortunately none of these patterns are supported.

A possibility to still get the required behaviour would be the use of Java Actions, which could start multiple threads, which then allows for asynchronous invocation of threads.

Pattern		Mendix	Explanation
12	Multiple instances without synchronization	-	Using a Java Action which starts multiple threads, this pattern could be realized. Using the looped activity with a variable specifying the number of instances and a micro flow inside the loop, this pattern could be serialized.
13	Multiple instances with <i>a priori</i> Design-Time knowledge	-	Same as for pattern 12, with added requirement for a parameter or variable specifying the number of instances.
14	Multiple instances with	-	Same as pattern 12, now only possible using a

	<i>a priori</i> Run-Time knowledge		variable.
15	Multiple instances without <i>a priori</i> run-time knowledge	-	Same as pattern 14.
34	Static partial join for multiple instances	-	Not supported, there is merely an exclusive merge available.
35	Cancelling partial join for multiple instances	-	Not supported, there is merely an exclusive merge available.
36	Dynamic partial join for multiple instances	-	Not supported, there is merely an exclusive merge available.

Table 24: Multiple instance patterns in Mendix

4.4.1 Pattern 12 – Multiple instances without synchronization

This pattern means that within a process, multiple instances of an action can be created which run independent and concurrently. This should also be without a requirement of synchronizing the instances afterward to comply with this pattern.

BiZZdesigner supports this pattern by using a replicated action which spawns of an AND-split, shown in Figure 38.

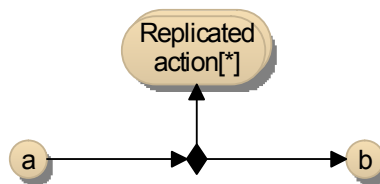


Figure 38: Pattern 12 in BiZZdesigner

4.4.2 Pattern 13 – Multiple instances with *a priori* design-time knowledge

This pattern is like the previous one, however now the amount of instances to be created is known and specifiable at design-time. It is now necessary to synchronize the instances before another activity can be triggered.

It is possible to create this type of behaviour in two ways, by using a replicated action with a specified number of instances. Just using an AND-split and AND-join with the same action multiple times in between would not be proper support of this pattern. This is shown in Figure 39, where an example is given for four instances of an action.



Figure 39: Pattern 13 in BiZZdesigner

4.4.3 Pattern 14 – Multiple instances with a priori run-time knowledge

Basically the same as pattern 13, however now the amount of instances is dependant on factors which are influenced at run-time, rather than known up front. As with pattern 13, the instances need to run independent and requires synchronization before other activities can be triggered.

There is no way to implement this behaviour directly, however using a counter (in an Item) with scripts one could realize it. In Figure 40, action a sets the number of instances in an item, b reads this item and creates the correct number of instances. Of course, there could be many other actions influencing the InstanceCounter.

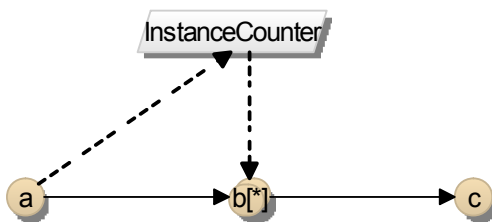


Figure 40: Pattern 14 with scripts required

4.4.4 Pattern 15 - Multiple instances without a priori run-time knowledge

Without a priori run-time knowledge, it is possible to initiate new instances of an activity at any time whilst the instances of the repeated activity are running. The instances need to be synchronized before being able to trigger other activities.

BiZZdesigner supports pattern 15 by use of a repeated action which checks if more instances are needed (Figure 41). After the number of instances properly created, it is synchronized with b, as all instances of b need to be done before it may trigger other activities (in this case action c).

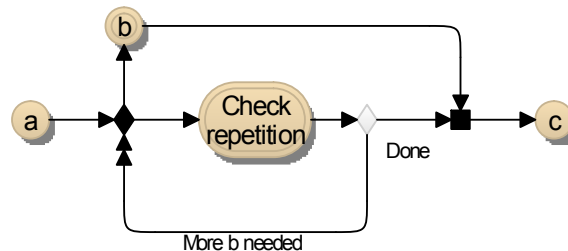


Figure 41: Pattern 15 in BiZZdesigner

4.4.5 Pattern 34 – Static partial join for multiple instances

A static partial join for multiple instances is similar to the structured partial join pattern (pattern 30), but now for joining a number of instances rather than actions. The join enables a following branch whenever a specified number of instances is complete, out of the total amount of instances.

There is no way to specify this in BiZZdesigner unless the instances are modelled as separate actions. However even then it would only be a construction like in Figure 29, which would get very cluttered in large amounts of instances.

4.4.6 Pattern 35 – Cancelling partial join for multiple instances

Basically the same as the cancelling partial join (pattern 32), but for instances rather than activities. The join enables the next branch as soon as the specified amount of instances is reached, and cancels the rest of the instances.

It should be clear that this pattern is not supported by BiZZdesigner, judging from the lack of support for patterns 32 and 34.

4.4.7 Pattern 36 – Dynamic partial join for multiple instances

This type of join is like pattern 34, however now the number of instances that is to be joined depends on runtime factors, rather than a priori knowledge.

As a static partial join (pattern 34) is already not supported in BiZZdesigner, it is clear that this one is not either.

4.5 State based patterns

State based patterns occur when the main determinant for the course of action that will be taken is the current state of the process.

Since there is not really a state concept in BiZZdesigner, most of the time “states” have to be stored in an Item to be able to make use of it. This means that only the deferred choice can be represented in BiZZdesigner, while lacking support for the other four patterns.

Pattern		BiZZdesigner	Explanation
16	Deferred choice	+	Can be realized using an AND-split to enable all outgoing branches, and having a disable relation after every first action of each branch to the other branches.
17	Interleaved parallel routing	-	Using an action with a script in it which keeps track of which branches have been active after an OR-split with no conditions. If all branches have been completed, the loop is done.
18	Milestone	-	There is no direct milestone construct, however using an action which checks the condition which is stored in a item and enables or disables based on that check
39	Critical section	-	No means of creating a kind of mutual exclusion
40	Interleaved routing	-	Even though not directly supported, using enumerating sequences it would be possible to simulate this behaviour.

Table 25: State based patterns in BiZZdesigner

The state based patterns in Mendix are all not available, due to the lack of support for parallelism. Another problem with these patterns is the lack of possibility for multiple triggers, which prevents simulation (as there is no direct support) of the milestone pattern.

Pattern		Mendix	Explanation
16	Deferred choice	-	This pattern requires two processes... etc
17	Interleaved parallel routing	-	No support for parallelism, so obviously this pattern is not supported.
18	Milestone	-	There is no direct milestone construct, nor a way to emulate this behaviour.

39	Critical section	-	Since there is no parallelism, a mutex type of construct is not present either.
40	Interleaved routing	-	Only enumerating every possible sequence and randomly choosing one would emulate this behaviour.

Table 26: State based patterns in Mendix

4.5.1 Pattern 16 – Deferred choice

The deferred choice is a point in the workflow process where one of several branches is chosen based on what branch is done processing first. The executed branch cancels execution of the other branches. Prior to the “decision” (since it is not a specifiable condition based on logic, it hardly is a decision) all outgoing arches from the deferred choice represent a possible path of execution.

In BiZZdesigner this pattern is realized by using an AND-split to enable all outgoing branches, followed by a disable relation going out from the first actions of those branches to the other branches. This means that whichever action gets done first, it will disable the other branches. In Figure 42, this behaviour is shown for two branches, where a disables the lower branch and c disables the upper branch, leading to either <Start, a, b, e> or <Start, c, d, e>.

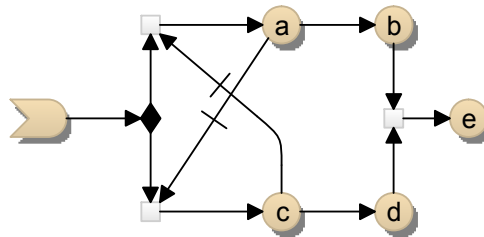


Figure 42: Pattern 16 – Deferred choice in BiZZdesigner

4.5.2 Pattern 17 – Interleaved parallel routing

If a set of tasks requires a partial ordering and is limited to execution once, the interleaved parallel routing pattern comes into play. It is used when there is a set of actions of which some need to be completed in a set order, but other than that can be randomly executed. It also requires that no two tasks can be active at the same time.

Even though not directly supported in BiZZdesigner, the behaviour can be enforced by using an action with a script in it which keeps track of which branches have been active after a random path

choice (an OR-split with no conditions). If all branches have been completed, the loop is done and the thread of control is passed on the next process entity.

4.5.3 Pattern 18 – Milestone

In the milestone pattern, an action is only enabled when the process is in a specific state. The milestone is a specific execution point in a process model, which can either enable the action or prevent it from ever happening. A milestone tests for a state and should not be a trigger.

Even though not directly supported by a milestone construct in BiZZdesigner, this could look like Figure 43. Here, an action checks the “state” which should be stored in an item (Milestone item in the figure). If at that point the milestone state is set to true, b is executed, otherwise that action is disabled and never executed.

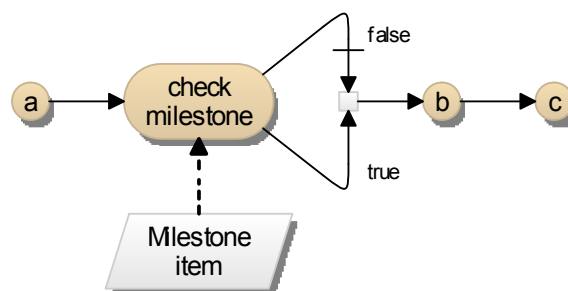


Figure 43: Pattern 18 – Milestone in BiZZdesigner

4.5.4 Pattern 39 – Critical section

The critical section pattern allows for specification of parts of a process which are to be executed non-concurrently. This means that if a process is executing in a critical section, other critical sections will have to wait for that one to finish.

There is no possibility to specify this type of behaviour in BiZZdesigner and Mendix.

4.5.5 Pattern 40 – Interleaved routing

Interleaved routing is about a set of activities that must be executed once and can be executed in a random order, but not concurrently. When all tasks are completed, the following task in the process has to be enabled.

Both BiZZdesigner and Mendix could only specify this type of behaviour by enumerating possible sequences and then explicitly select a path. This is not proper support of the pattern.

4.6 Cancellation and force completion patterns

In this type of patterns, currently active activities are cancelled. BiZZdesigner offers support for one of this type of patterns, the disable relation would seem to perform this kind of behaviour at first sight.

Pattern		BiZZdesigner	Explanation
19	Cancel activity	+	The disable relation supports this pattern (conform the first variant of the pattern as in figure 25 of [RHAM06]), however aborting a currently executing action is not possible as actions are seen as atomic.
20	Cancel case	-	No cancellation feature is available to remove an entire process instance.
25	Cancel region	-	There is no cancellation feature for regions, the only possibility would be to add disable relations to all activities in the region you would want to disable.
26	Cancel multiple instance activity	-	No cancellation feature for multiple instances.
27	Complete multiple instance activity	-	No forced completion feature available.

Table 27: Basic cancellation patterns in BiZZdesigner

Cancel activities in Mendix are not directly available, only using java actions could these be performed. Since we disregard this, we consider the patterns not supported.

Pattern		Mendix	Explanation
19	Cancel activity	-	Cancelling running activities is only possible if an error occurs.
20	Cancel case	-	Mendix has a standard rollback in case an error occurs, which could be seen as a “cancel case”. However, this functionality is only forced on a error rather than having a construct to do a proper cancel case.
25	Cancel region	-	Mendix has a standard rollback in case an error occurs, forcing an error at the end of a region might force this behaviour if you set custom error handling before the region you want cancelled. However, a proper construct for this is not available.
26	Cancel multiple instance	-	No ability to run multiple instances concurrently,

	activity		as well as no direct cancel activity lead to no support of this pattern.
27	Complete multiple instance activity	-	This pattern requires patterns 13-15, which already are not supported.

Table 28: Basic cancellation patterns in Mendix

4.6.1 Pattern 19 – Cancel activity

A task is cancelled prior to, or during its execution by this pattern.

BiZZdesigner supports this pattern in the form of a disable relation, which disables actions prior to their execution. Since actions are seen as atomic, disabling during execution is impossible.

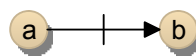


Figure 44: Pattern 19 – Cancel activity in BiZZdesigner

4.6.2 Pattern 20 – Cancel case

Cancel case removes an entire process instance, including currently executing tasks, tasks that may be activated in the future and sub-processes.

BiZZdesigner does not directly support this pattern. The only option would be to include disable relations at every action coming from an AND-split enabled by a cancel trigger or activity which initiates the cancellation of the case.

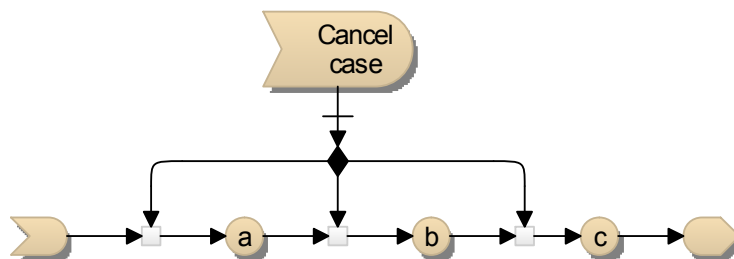


Figure 45: Pattern 20 – Cancel case in BiZZdesigner

For both tools, all of the advanced cancellation patterns are not available.

4.6.3 Pattern 25 – Cancel region

The cancel region pattern disables a set of tasks in a process instance. If they are currently being executed they are withdrawn. The tasks that are to be cancelled do not have to be a connected sub set of the process.

There is no cancellation feature for regions in BiZZdesigner, the only possibility would be to add disable relations to all activities in the region you would want to disable, which is a solution similar to the one given at pattern 20.

4.6.4 Pattern 26 – Cancel multiple instance activity

Cancels independently running instances of a task that is running. Instances that are already complete are unaffected by this pattern, but those that have not finished (completely) are withdrawn.

This is not supported in BiZZdesigner, it is only possible to cancel a replicated activity completely before execution of it is started.

4.6.5 Pattern 27 – Complete multiple instance activity

Basically the same as pattern 26, but now the tasks are forcibly completed, meaning that subsequent actions can be performed.

BiZZdesigner does not support this pattern, as there is no way of forcing completion present.

4.7 Trigger patterns

Trigger patterns handle external signals which are aimed at the start / enabling of tasks. In BiZZdesigner there is a start- and endtrigger, of which only the start trigger is relevant to these patterns.

Pattern		BiZZdesigner	Explanation
23	Transient trigger	-	If a trigger is activated, this is durable (see pattern 24).
24	Persistent trigger	+	The first variant is just a trigger, the second is modelled as an extra input trigger and an AND-join which ensures continuation of the process if the trigger is active and the process is ready for it.

Table 29: Trigger patterns in BiZZdesigner

Triggers in Mendix are always in the form of a single start and end event. This means that the transient trigger and persistent trigger patterns are not supported.

Pattern		Mendix	Explanation
23	Transient trigger	-	A start event is always durable.
24	Persistent trigger	-	Since only one start event is possible in a micro flow, there is no option to add persistent triggers.

Table 30: Trigger patterns in Mendix

4.7.1 Pattern 23 – Transient trigger

A transient trigger sends an enable which is lost if not directly acted upon by a receiving task. This means the trigger can only be used if there is a process waiting for it.

In BiZZdesigner, triggers are durable in nature, which means this pattern is not supported.

4.7.2 Pattern 24 – Persistent trigger

Persistent triggers are a signal from the external environment which are retained by the process until they can be acted upon by a receiving task. It is the counterpart of the transient trigger, in the sense that it is durable in nature.

BiZZdesigner supports persistent triggers, if used with an AND-join the process can get blocked until it receives the trigger (much like a timer for example), as shown in Figure 46. Possible paths are:

<start, a, b> (PT never activated)

<start, a, b, PT, c, end>

<start, a, PT, b, c, end>

<start, PT, a, b, c, end>

<PT, start, a, b, c, end>

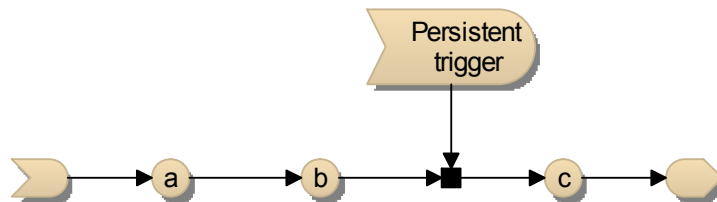
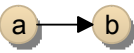
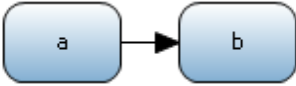
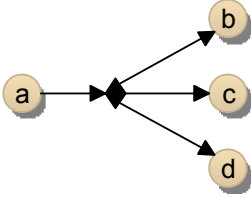
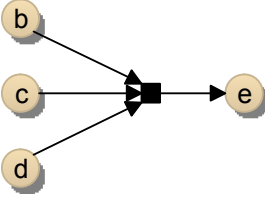
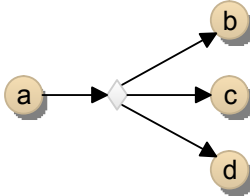
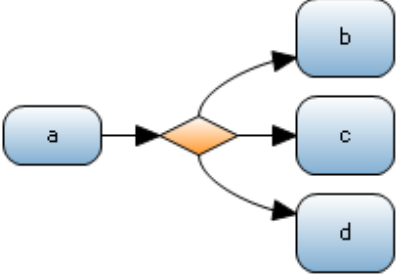
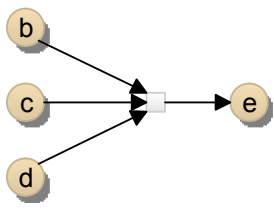
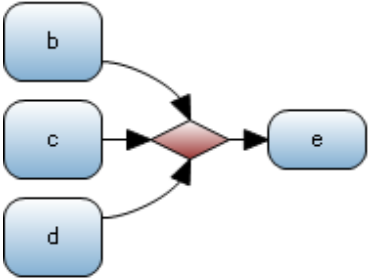
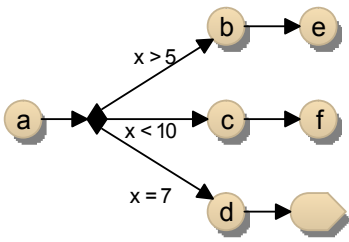
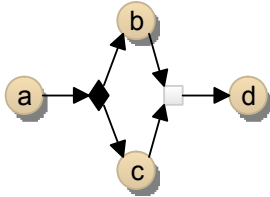
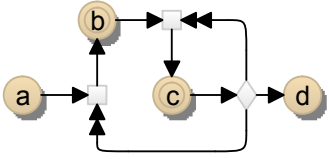
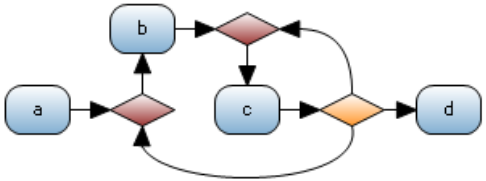
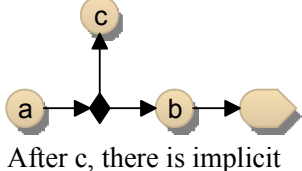


Figure 46: Pattern 24 in BiZZdesigner

4.8 Overview

Now, an overview will be given of found mappings to the patterns by both BiZZdesigner and Mendix, along with problems when creating a mapping. The problems are isolated to the patterns which Mendix fails to comply with, while BiZZdesigner provides support for it. The outcome of this analysis will be used in chapter 5 to establish guidelines for modelling.

Pattern		BiZZdesigner	Mendix
1	Sequence		
2	Parallel split		-
3	Synchronization		-
4	Exclusive choice		

5	Simple merge		
6	Multi-choice		-
7	Structured synchronizing merge	-	-
8	Multi-merge	-	-
9	Structured discriminator		-
10	Arbitrary cycles		
11	Implicit termination		-

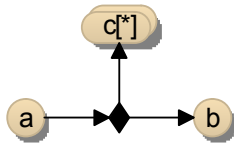

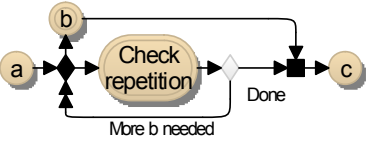
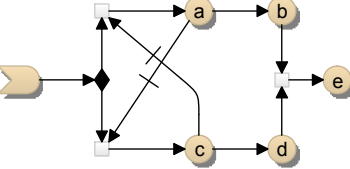
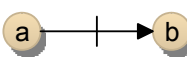
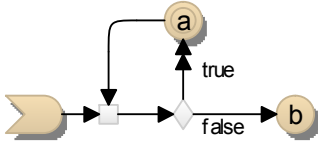
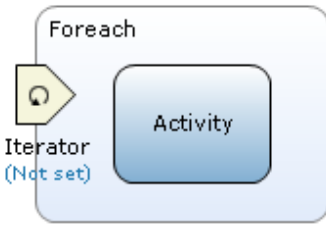
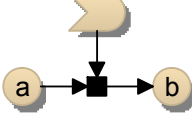
		termination	
12	Multiple instances without synchronization	 <p>action c is replicated, and is not synchronized with the process afterwards</p>	-
13	Multiple instances with <i>a priori</i> Design-Time knowledge	 <p>Four instances of b</p>	-
14	Multiple instances with <i>a priori</i> Run-Time knowledge	-	-
15	Multiple instances without <i>a priori</i> run-time knowledge		-
16	Deferred choice		-
17	Interleaved parallel routing	-	-
18	Milestone	-	-
19	Cancel activity		-
20	Cancel case	-	-

Table 31: Pattern support for both tools (part 1)

Pattern		BiZZdesigner	Mendix
21	Structured loop	 <p>action a is repeated until the condition in the split is false</p>	 <p>The activity is repeated until the condition in the foreach is reached</p>
22	Recursion	-	-
23	Transient trigger	-	-
24	Persistent trigger	 <p>The process is halted until the persistent trigger is activated</p>	-
25	Cancel region	-	-
26	Cancel multiple instance activity	-	-
27	Complete multiple instance activity	-	-
28	Blocking discriminator	-	-
29	Cancelling discriminator	-	-
30	Structured partial join	-	-
31	Blocking partial join	-	-
32	Cancelling partial join	-	-
33	Generalized AND-join	-	-
34	Static partial join for multiple instances	-	-




35	Cancelling partial join for multiple instances	-	-
36	Dynamic partial join for multiple instances	-	-
37	Local synchronizing merge	-	-
38	General synchronizing merge	-	-
39	Critical section	-	-
40	Interleaved routing	-	-
41	Thread merge		-
42	Thread split		-
43	Explicit termination	-	

Table 32: Pattern support for both tools (part 2: advanced)

4.9 Conclusion

This chapter focussed on the behavioural capabilities of both modelling languages, showing weaknesses and strengths on various types of workflow patterns. The languages were quite far apart in their ability to match the patterns, even over just the first 20 patterns. This is clearly visible in Table 31, where it is shown that BiZZdesigner was able to represent 14 out of 20 whereas Mendix was stuck at 4. Of particular interest was the fact that Mendix microflows are unable to handle parallel processes. This is a major issue considering the effect the absence of that pattern has on a lot of other patterns. Both languages performed rather poor at representing the more advanced patterns (21 to 43, Table 32), where BiZZdesigner scored 4 out of 23, and Mendix 2 out of 23.

Furthermore, it can be observed that of the 20 patterns that BiZZdesigner can represent, only 5 can be matched to Mendix microflows. The lack of behavioural capabilities of Mendix has several implications for a mapping which shall be discussed in chapter 5.

5 GUIDELINES FOR MODELLING

The previous chapters have determined a number of differences between the representational and behavioural capabilities of the BiZZdesigner and Mendix languages. Now that these matches and gaps are clear, it is time to convert them into guidelines for modelling in BiZZdesigner in order to answer the final research sub-question about methodological support for a practical mapping. This chapter will cover the limitations for construct use, the need for extra information to be added, as well as document what kinds of information get lost during a mapping. By doing so, this chapter will cover “dos and don’ts” when modelling something in BiZZdesigner for a mapping to Mendix. An example model will be used to illustrate the problems and solutions that arise during a typical process model mapping.

5.1 BWV and Control flow issues

The BWV analysis of chapter 3 established problems in the area of various constructs, being items, actors and blocks. There were also issues concerning the operations on items and lawful transformations such as the AND-split. Lastly, there was some excess in the form of the repeated relation, repeated action and the replicated action. In chapter 4 some of the same and other problems were established, such as problems with (end)triggers and disable relations. All of these issues will now be discussed and be used to form guidelines for modellers.

5.1.1 Issue 1: parallelism and synchronization

Judging from various patterns in chapter 4, especially patterns 2, 3 and 6, it is obvious that there is no parallelism possible in Mendix. BiZZdesigner models on the other hand make a lot of use of these patterns in the form of the AND-splits and -joins.

In an article about concurrent operations, Suleiman et al. also mention that even though processes may be concurrent, they will always be executed by the server serially [SCF97]. This can lead to problems, where serialized behaviour is different from the intended concurrent behaviour. Serializing in a process model would seem to generate the same kind of problem. Take for example the process in Figure 47, where managers x and y need to check the result from action a once a week. The managers both only have time for this once per week, at varying times. If serialized, the process would look like: <start, a, x, y, end> or <start, a, y, x, end>. If you would pick the first variant, but manager y only has time on Monday and manager x on Friday, the process cannot be completed for that week. An example for the second variant is of course alike. Of course, there would be a possibility to add a “check agenda” activity to decide on the order, however after this check activity something in the agenda could change, leaving us with the same problem.

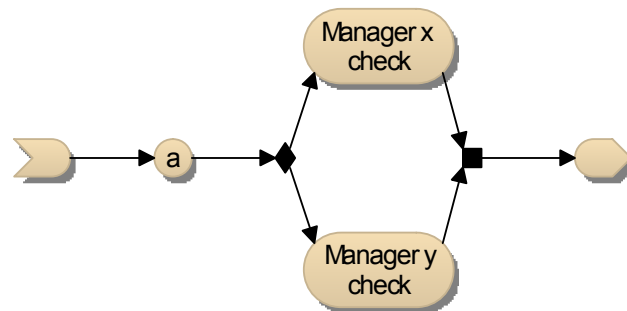


Figure 47: Example parallel process

Since Mendix will cannot represent parallel concept, an option is to serialize the process, possibly with an added (random) path choice mechanism to somewhat emulate the behaviour (the choice would be between all possible paths that can be followed if an AND-split were to be used). As said, this solution is not always a proper representation of intended behaviour and could also lead to rather large models.

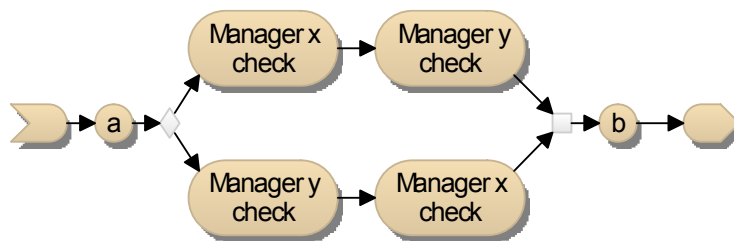


Figure 48: Possible paths of figure 48

Another option would be to make use of custom built java classes in Mendix which could call the parallel branches. This would mean that the AND-split would get mapped to a Mendix Java action, and that the branches need to be in a separate microflow.



Figure 49: AND-split by Java action in Mendix

In this particular case, the AND-split handler would be written to call two newly created microflows. One would only be containing a start event, the “Manager x check” activity and an end event while the other would contain the check for manager y in the same representation. Of course, when considering other models these flows could be more complex.



Figure 50: Microflows that are called by the Java action in Mendix

Guideline 1: And-splits and And-joins should be used sparsely. If possible, the modeller should determine the order of otherwise parallel modelled activities.

5.1.2 Issue 2: replicated actions

BiZZdesigner's replicated actions are the start of multiple instances (patterns 12 to 15) of the same action within a process. The replicated action was also noted in chapter 3, where it was already noted as excess without a Mendix equivalent. It would seem Mendix does not have a similar concept for this, as well as no possibility to specify the number of actions separately coupled by an AND-join and split.

The only solutions seem to be to disallow the use of replicated actions, or to serialize the number of required actions (which makes it a repeated action, rather than replicated), which would lead to the situation described at issue 1. Another option would be to implement a custom java class and use it in a microflow in Mendix to perform the modelled behaviour.

Guideline 2: Do not use replicated actions.

5.1.3 Issue 3: start and endtriggers

From the different results in patterns 11, 23 and 24 there are two observations that can be made, the first being the requirement for an explicit endpoint in Mendix while BiZZdesigner uses implicit ones. Determining where a process ends and adding an end trigger there should be a possibility for already existing BiZZdesigner models, as the simulator in BiZZdesigner is already able to do so at this point in time. For newly created models, it may be easier to require an endtrigger as a last “action” in a process. The addition of the construct will not change the BiZZdesigner model in terms of behaviour.

Guideline 3: Use endtriggers in BiZZdesigner whenever a process is supposed to end, avoid implicit ends.

The second observation is the presence of only one start trigger in Mendix processes, while there can be multiple in BiZZdesigner. Solving this is probably not easy, an option may be to add a parameter for each start trigger in BiZZdesigner, which then decides where to start the process in Mendix (or which micro flow to start, if modelled separately). However, this is not possible if multiple triggers are enabled within the same process model, leading to the solution of prohibiting more triggers in BiZZdesigner.

Guideline 4: Only use one trigger in BiZZdesigner models. Split up the model if there is a need for more than one.

5.1.4 Issue 4: differences in relations

Based on observations of differences in the ontology of both tools (chapter 3) and the different results for patterns 10, 19 and 21, there are two problems that can be identified in the relations of BiZZdesigner compared to microflows.

Firstly, there is no counterpart for a disable relation in Mendix. The disable relation is mostly used in combination with parallelism to disable another branch if a certain activity has been completed. Since parallelism cannot be used it seems feasible to prohibit use of the disable relation. BiZZdesigner has indicated that this relation is not used frequently, which makes it even less of a problem.

Guideline 5: Do not use the disable relation.

Secondly, there is the repeated relation in BiZZdesigner which is not explicit in Mendix. The fact that the repetition of actions is not explicitly visible in Mendix is not a problem for the mapping. It seems to be a purely visual discrepancy, rather than it having any problematic implications.

Guideline 6: Use normal relations rather than repeated relations for clarity.

5.1.5 Issue 5: items and item operations

Items in BiZZdesigner can represent real-life things such as letters and invoices. This is impossible to represent in Mendix, which means they should be limited to items that have a datatype.

Guideline 7: Do not model real life things, only model items with an appropriate datatype.

Operations on Items are shown quite different in BiZZdesigner compared to Mendix. Mendix requires a separate action which performs the operation, while BiZZdesigner just shows a link to an Item with the operation specified in the link. If mapped, automatically adding the Mendix action type should probably be possible.

Guideline 8: For clarity, add a separate action in BiZZdesigner that performs the CRUD operation if an action performs it.

5.1.6 Issue 6: repeated action

The repeated action in BiZZdesigner basically performs the same action a number of times. If an action is labelled repeated in a loop this is no problem to map to Mendix, as it has no further implications (construct excess). However, if it is noted to have a specified amount of times Mendix should perform that action the specified number of times in a sequence, which could get quite large if the number is high. This can be solved by use of the microflow action call, which calls a microflow containing the number of activities.

Guideline 9: Only use the repeated action when in a loop. If it is needed to perform the same action a number of times, create a sequence.

5.1.7 Issue 7: blocks

Blocks are a fairly large problem described in the BWW chapter (3), which has no Mendix equivalent. An option would be “flattening” the BiZZdesigner model, meaning creation of one large model without any layering. However, considering the scale of some models this is probably not appropriate. To match the blocks, it should be possible to create a microflow for every block and calling those microflows in a “parent” microflow.

Guideline 10: Only use blocks to separate activities if the content can be seen as a separate process.

Blocks come with an additional challenge, as there is a difference between BiZZdesigner and Mendix endtriggers. If mapped directly, an endtrigger in a BiZZdesigner block will not end the main process of the resulting Mendix model as would be intended. This can be observed from the problems with

patters 22 and 43 of the control flow analysis. For this reason, endtriggers should only be placed at the main process level, unless it is the intention that the main process continues afterwards.

Guideline 11: Do not use endtriggers within blocks, unless the main process should continue.

5.2 Granularity and modelling level

It would seem that there are quite some limitations to a mapping from BiZZdesigner to Mendix. This could be caused by a difference which is more of a conceptual problem [BAU08] than just a number of language problems. It is likely for a difference to occur, as for example Dehnert and van der Aalst [DA04] argue, with business process models and a workflow specification. Differences occur because of different goals that are pursued, being intuitive communication at the business process level and a more rigorous, software interpretable description for the workflow. However, both types of models still cover the same area of interest, being the performed tasks and their order. The same type of situation is at hand for BiZZdesigner and Mendix, this section will assess the modelling levels of both tools to see how large the gap is that needs to be bridged.

5.2.1 BiZZdesigner modelling level

In zur Muehlen's work about workflow based process controlling [MUE04], a business process and a workflow are on a different level of abstraction. A process in general is *"a discrete, holistic, temporal and logical sequence of activities which are needed to manipulate an economically relevant object"*. The business process is defined as a high level process, determined by the overall goals of an enterprise. The business process contains information about activities that interface with partners like customers, suppliers, etcetera. The BiZZdesigner models are clearly on this level, which is conform BiZZdesign's description of BiZZdesigner as *"a tool for business process modelling, used for designing, documenting and communicating processes, organisation, information, working instructions and related documentation"*.

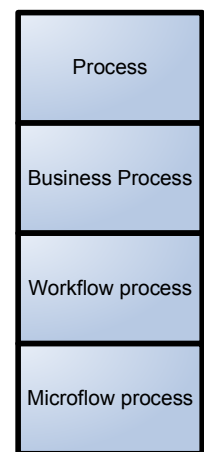


Figure 51: Overview of processes

5.2.2 Mendix modelling level

According to Muehlen [MUE04], a workflow is a specific representation of a business process, which allows for formal coordination between activities and applications. In a workflow model, the process participants can be controlled by a workflow management system. Mendix is currently trying to get onto this level of modelling, but seems to be at a lower level at this point in time with microflows. This is because non-system tasks are not covered (aside from the forms), as well as a lack of specification for actors that need to perform tasks.

Guideline 12: Model system tasks rather than human tasks.

Although Mendix works purely with models instead of a textual programming language, their focus on details of software development make it more of a graphical programming language rather than a workflow specification. Even though the BPMN-like notation would make one expect differently, the level of detail between a business process and the microflows are a bridge too far in terms of a direct mapping. Figure 51 (adapted from [MUE04]) illustrates this, going from BiZZdesigner to Mendix would not only mean covering the gaps from business process to a workflow specification, but going to a lower level. A direct mapping would imply skipping more than one level of abstraction, leaving the process designer with a lot of required input.

Guideline 13: Use detailed tasks, but also use layering to keep the top-level model on the business level.

This guideline is somewhat conflicting with guideline 10, which noted that blocks should only be used for separate processes. However, we assume that a high-level business process action can also be seen as a separate (sub-) process for the top-level business process model. This means that if an action is not detailed enough at first, the actions that are added to enable a more direct mapping to software are modelled within a block that has the name of the original action.

5.3 Loss of information

When mapping from BiZZdesigner to Mendix, some information will be lost. The information that will get lost is important, since it may very well affect the desired outcome in Mendix. The information that gets lost is shown in Table 33 below. Business process modellers should be aware of them, and be sure to find a way aside the model to communicate this lost information.

Loss concern	Constructs	Explanation
Probabilities and choice expressions	AND- and OR-splits	The expressions and probabilities that determine the choice will have to be manually transferred, as these cannot be automatically mapped to Mendix expressions. Probabilities are especially of importance when the AND-split is concerned, as this means that the split is not always activating all outgoing arches.
Item operations	Items	Items currently do not get a direct mapping, although it would be possible to map the item operations fairly straightforward to object actions in Mendix. However, it is quite usual that an action in BiZZdesigner not only operates on the Item, but also requires some user input, etcetera.

Repeated relation and actions	Repeated relation / actions	Since the repeated relation is getting mapped to a normal flow just like an enable relation, it is impossible to make a distinction after the mapping. This also goes for the actions involved in this repetition (see arbitrary cycle pattern), which get mapped to standard actions in Mendix.
Actors	Actors and actions	Mendix is unable to link actors to actions that they perform, this information will get lost in a direct mapping

Table 33: Information losses

After mapping a business process model, the resulting model needs to be properly refined in order to come to an executable version of the model.

5.4 Example Mapping

To illustrate the proposed mappings and the issues that have been discussed in the various previous chapters, an example process will now be mapped from BiZZdesigner to Mendix. The process will be a typical BiZZdesign process called Pro-fit, of which a part was already shown in paragraph 2.3.2. Pro-fit is an imaginary insurance company whose business process model consists out of four main parts, being registration, acceptance, examination and payment. For purposes of this example, the model was slightly altered to show how to use the guidelines in practice as much as possible. In this chapter the BiZZdesigner Items are all related to a datatype, in correspondence to guideline 7. Besides that, every item operation will require the creation of a separate activity (guideline 8) to perform the desired operation to match Mendix' format of activity types. This will only be explained once to keep the model from getting cluttered with item operations.



Figure 52: Pro-fit claim settlement blocks

An initial mapping of the top level process is fairly easy, as shown in Figure 53. During this chapter, this top-level process will be updated and presented where needed.

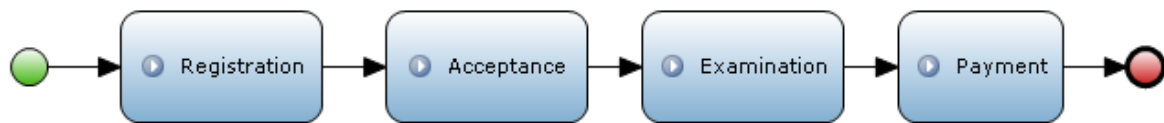


Figure 53: Initial mapping of the Pro-fit settlement blocks in Mendix

5.4.1 Registration

Starting with the Registration block, which in short checks whether or not a customer exists, creates files for the claim and checks if everything that is required is present.

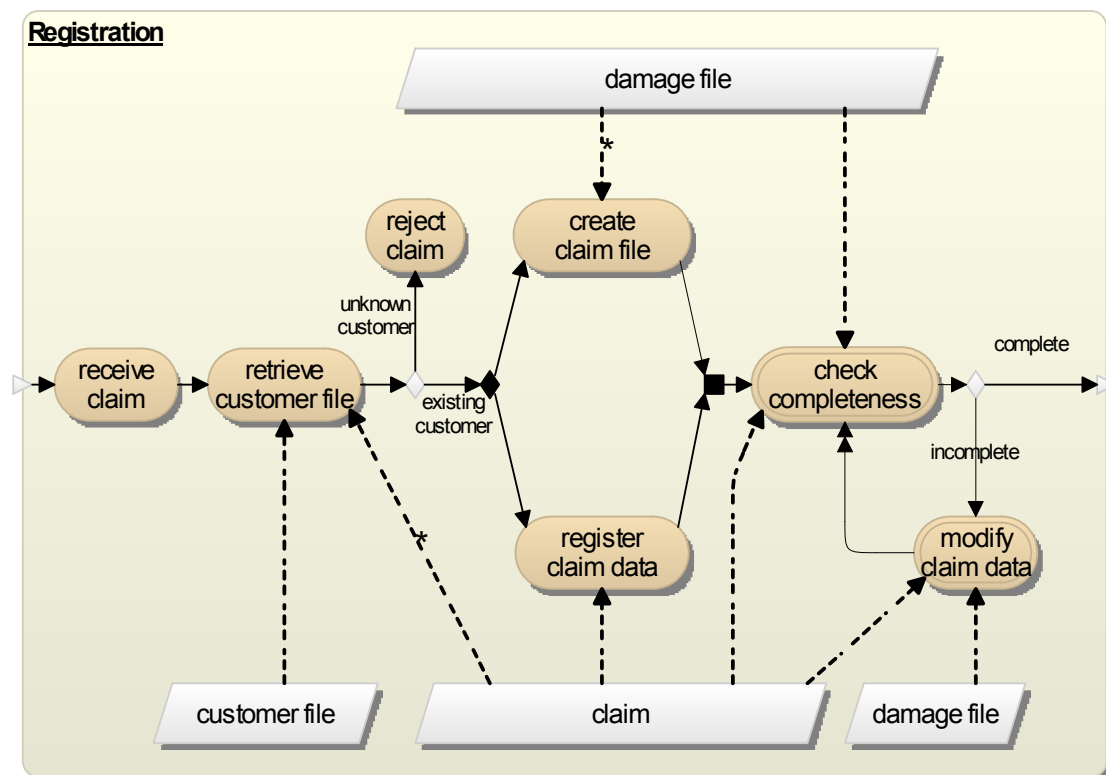


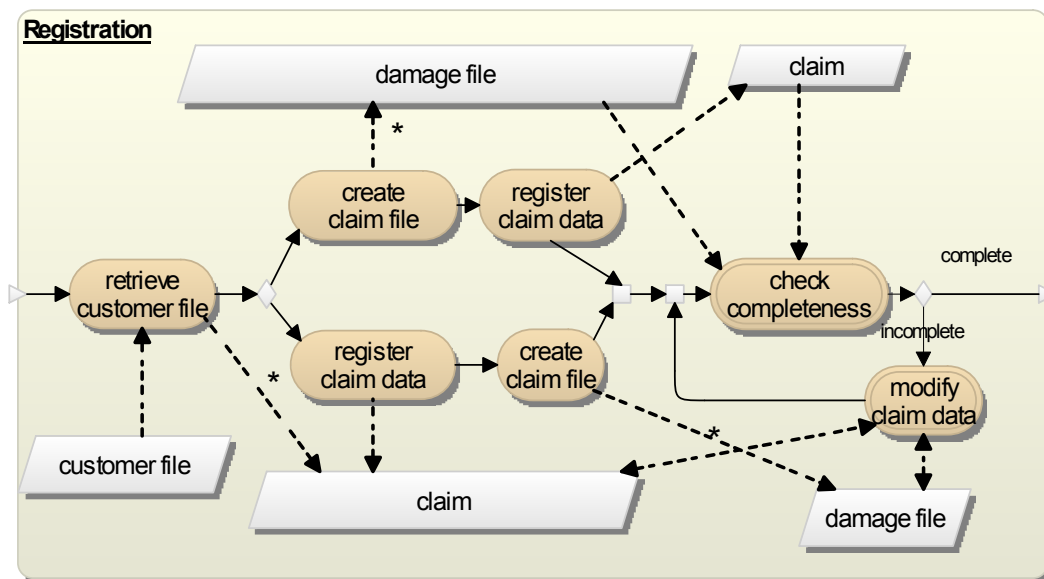
Figure 54: Registration process block in BiZZdesigner

The first problem that can be observed is the lack of an endtrigger after the reject claim activity. The initiate solution (guideline 3) would be to add an endtrigger after this activity, however this causes

issues with Mendix returning to the main process and continuing with the acceptance process even though it should end. In order to create the proper behaviour in Mendix, the decision will have to be taken at the main process level rather than within a block as specified in guideline 11. In this situation, we choose to extract the receive claim and the rejection choice and place it before the registration block.

The second issue is the AND-split that triggers the activities “create claim file” and “register claim data”. Using guideline 1, the possible sequences are to be created and placed after an OR-split.

The repeated actions “check completeness” and “modify claim data” are fine, in compliance to guideline 9. However, the repeated relation should be replaced by a normal one in order to comply with guideline 6. The resulting BiZZdesigner process is shown in Figure 55, which can then be mapped to the Mendix microflow of Figure 56.



F

Figure 55: Registration BiZZdesigner block after adjustments

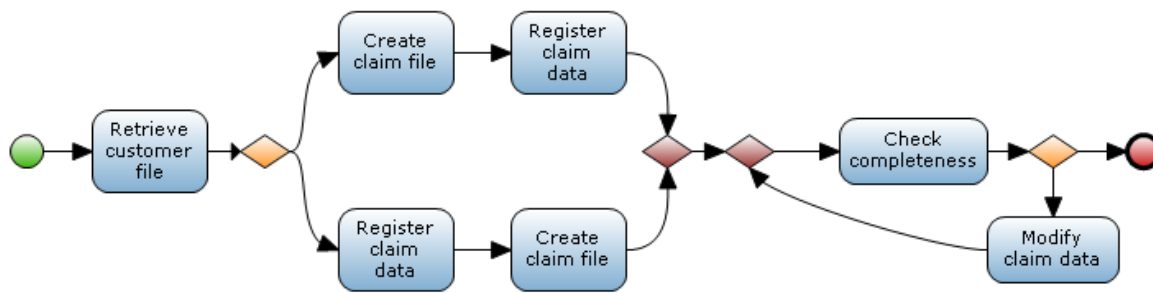


Figure 56: Registration microflow in Mendix after adjustments

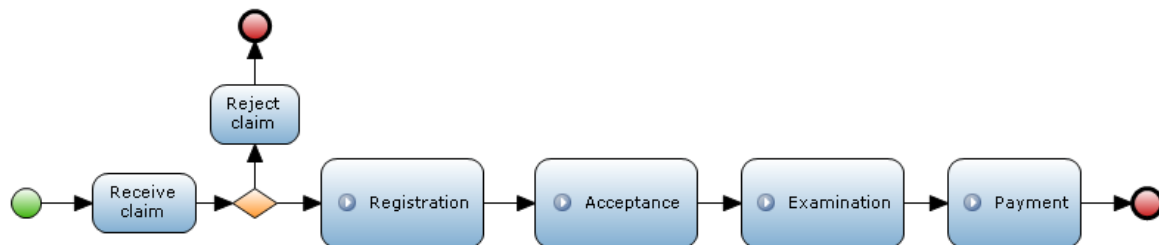


Figure 57: New main process in Mendix

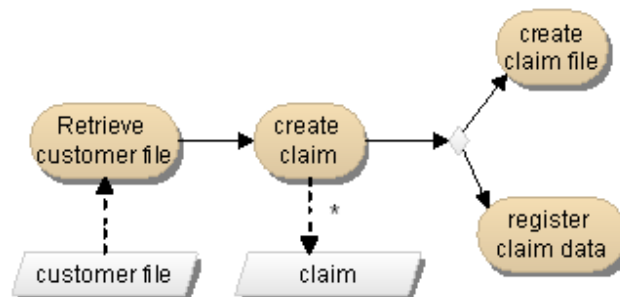


Figure 58: Retrieve customer file item operations

Now it is time to show guideline 8, the result of which is shown in Figure 58. The retrieve customer file action is split up into two parts. The first action is the action that actually retrieves the customer file, which means it is not needed to add another action for the originating action. The second action creates the claim item, after which the process continues in the same way as before.

5.4.2 Acceptance

The acceptance block is about whether or not a claim is susceptible for compensation by the insurance company. Data on the insurance form is checked and altered where needed, followed by a decision on acceptance.

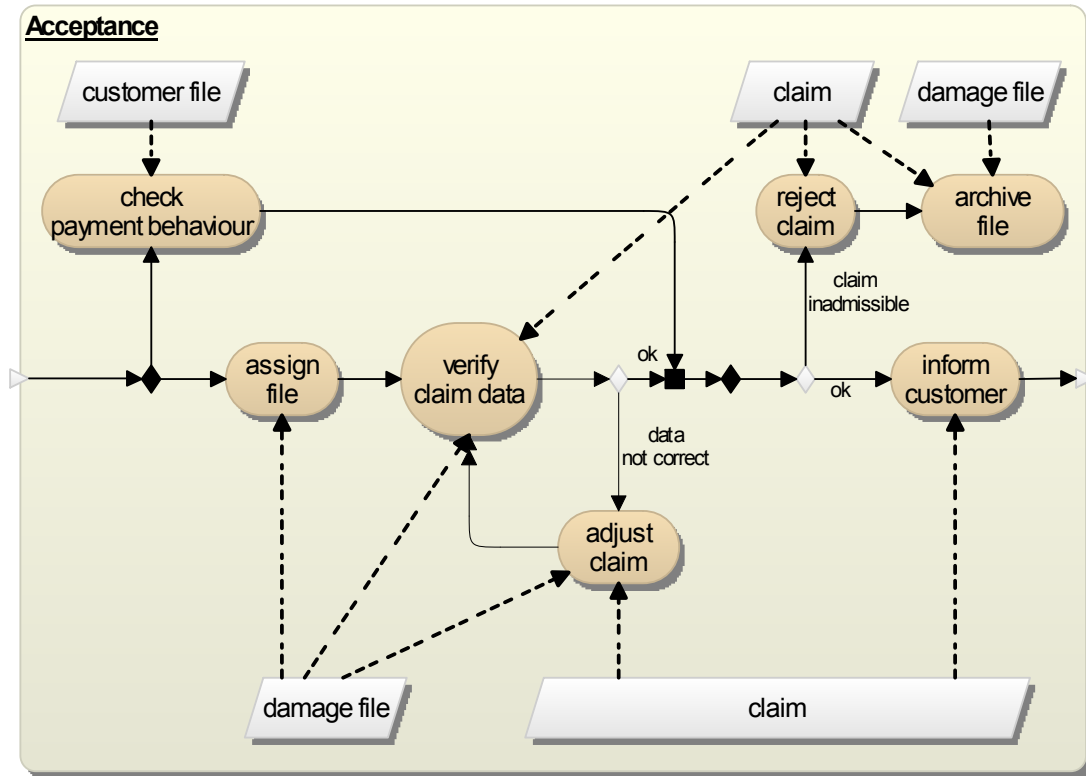


Figure 59: Acceptance process block in BiZZdesigner

The acceptance block starts off with another AND-split problem, which needs rewriting in order to be mapable to Mendix. This time around, applying guideline 1 is more complicated compared to 5.4.1, as there are more activities before the AND-join. Just enumerating the possibilities is an option, however this is complicated by the OR-split with the activities that verify the correctness of the claim data. In this case, as a modeller we presume that it is realistic that assigning a file and checking payment behaviour will take less time than verifying the claim data. We therefore create the sequence that is to be expected in this scenario. This is not the exact same behaviour as before, but the lack of parallelism is not expected to create a long waiting period for the claim data verification.

Another problem that is similar to that in 5.4.1 is the lack of an endtrigger at the top branch of the admissibility check OR-split. The process should end after the “archive file” activity. Applying guideline 13, this means this decision will have to be taken at the main process level rather than within this block. However, we presume that a customer needs to be informed of the claim's inadmissibility. This means that it is safe to add an OR-join before the “inform customer” that is connected with “archive file” and the “ok” arrow from the admissibility OR-split. However, this also requires a split after the acceptance block to make sure that the process is not continued if a claim was found inadmissible. The resulting block is shown in Figure 60, the Mendix equivalent in Figure 61.

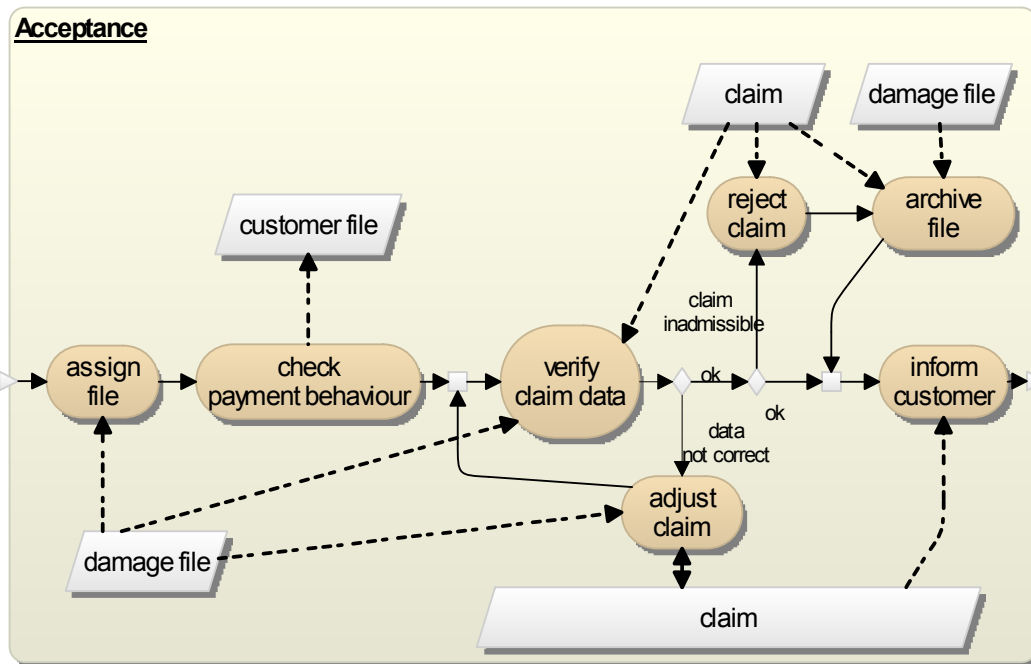


Figure 60: Acceptance block after adjustments

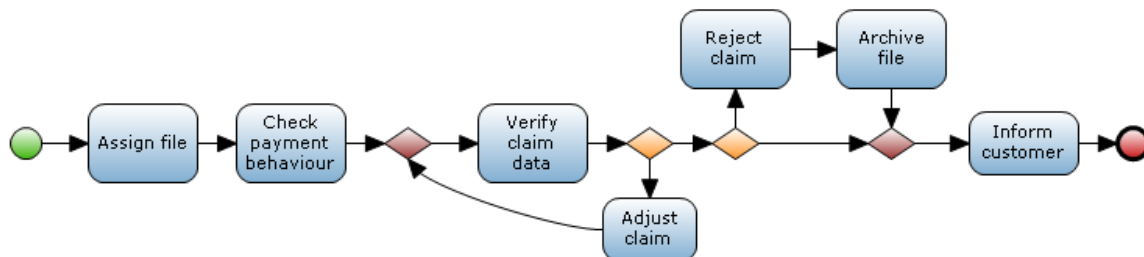


Figure 61: Acceptance microflow after adjustments

5.4.3 Examination

The examination block is about deciding on full or partial payment, potentially consulting a specialist and re-assessing the damage. The decision that is taken is used at the payment block which is discussed in the next section.

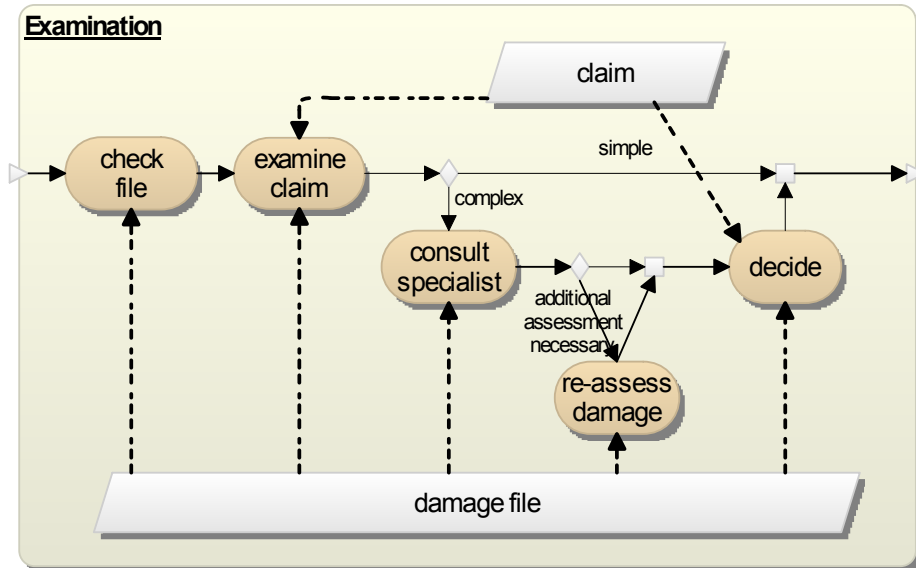


Figure 62: Examination process block in BiZZdesigner

This block has a lot of human tasks in the process, these tasks need to be redesigned to comply with guideline 12. For example, the “examine claim” is done purely by a damage specialist based on the claim and damage file. The retrieval of information and showing this information to the specialist is what the system is doing, rather than examining the claim (which is the human task). Based on the input that the specialist provides, the process continues as simple or complex. This is required if the process is intended to be executed as quickly as possible after the mapping. For now however, the business process block can already be mapped directly onto Mendix microflows (Figure 63) since there are no problems with the BiZZdesigner constructs that are used.

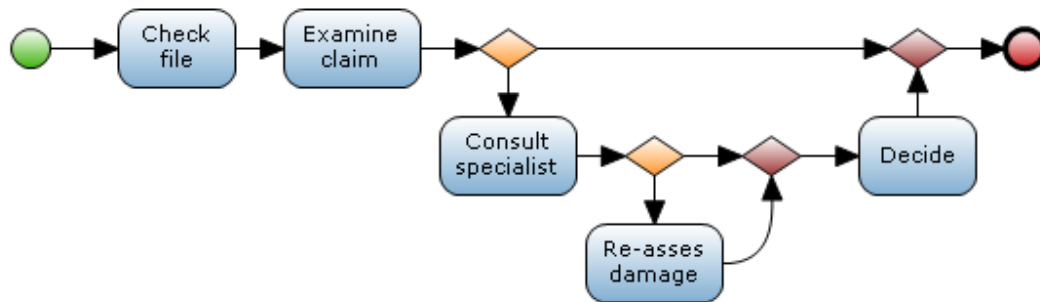


Figure 63: Examination microflow in Mendix after adjustments

5.4.4 Payment

The final block, Payment, is about partial or full payment of the claimed amount. Even if the claim was not admissible, the customer is informed of the decision and the damage file will be archived.

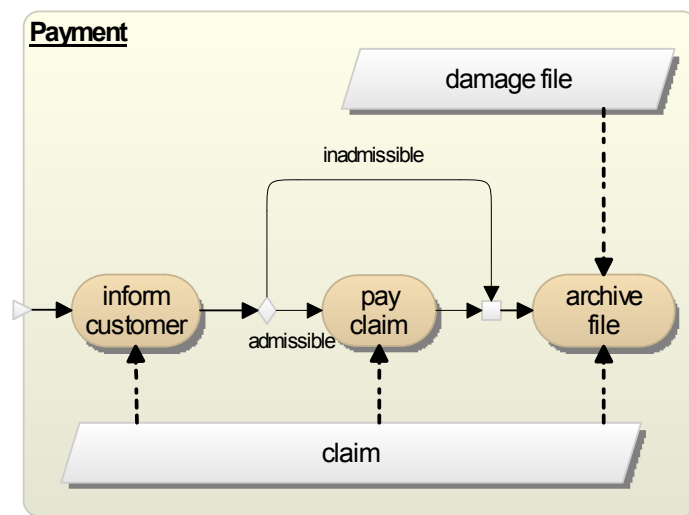


Figure 64: Payment process block in BiZZdesigner

This block is fairly small and simple but lacks an endtrigger or exit out of this block. Other than that, this process is ready for a mapping to Mendix. Since the addition of the endtrigger is trivial, there is no need to provide another figure containing this small change in BiZZdesigner. The corresponding Mendix microflow is shown in Figure 65.



Figure 65: Payment microflow in Mendix

6 CONCLUSION AND DISCUSSION

In the previous chapters of this thesis, the mapping from BiZZdesigner to Mendix was discussed extensively. The languages were described and compared by use of an ontological and behaviour based analysis to establish mapping possibilities and problems. Suggestions to solve or workaround some of these problems were proposed in both chapters 3 and 4. Then, general guidelines for modelling in BiZZdesigner were proposed and an example process was used to illustrate problems when going from a BPM-oriented process modeller to a modeller designed for software development. This final chapter will give an overview of the thesis' results, along with its relevance, limitations, suggestions for future research and recommendations.

6.1 Summary and conclusion

At the start of this document, this thesis' main research question was presented as: *"How to transform the BiZZdesigner language into the Mendix microflow modelling language?"*. To answer this question, the first step was to do a literature study of relevant research material. This literature study pointed out that the best way to start with creation of a mapping is to first find out the capabilities of both languages using an ontological and a behavioural analysis. The main research question was split up into four parts, being:

- What are the semantic differences between the two languages?
- What is the difference in expressive power of the two languages?
- How to define a mapping between the two languages in order to support software development?
- How to methodologically support a practical mapping between the two languages?

The BWW ontological analysis was used to answer the first sub-research question of this thesis, which was to find out semantic differences between the two languages. The ontological analysis of chapter 3 had several interesting findings, most importantly the lack of representation for real-life things in Mendix, which also influences its capacity for system representation. Besides that, a lack of support for system decomposition and differences in lawful transformations with regard to parallelism was found. An overview of capability matches and issues was provided as basis for a mapping from BiZZdesigner to Mendix in section 3.3.

To answer the second research question, a control flow pattern analysis was used to find differences in expressive power of the two languages. This analysis covered forty-three commonly used behaviour patterns. Both languages were measured up to these patterns, which lead to a table of differences in control flow support (see section 4.8). This was of interest as specified behaviour in BiZZdesigner should be representable in Mendix for a proper mapping. The most striking problem was the lack of support for parallelism in Mendix, which meant quite a few patterns were not supported. A table with matching patterns was provided for the mapping of behaviour in section 4.8, going from the BiZZdesigner language to Mendix. Together with the last section of chapter 3, this answers our third research question.

After analyzing all the matches and problems, the knowledge of these gaps and possible workarounds was turned into guidelines for modelling for Mendix in BiZZdesigner at chapter 5. Along with an example to show the practical impact of the mapping issues, differences at modelling level were described and guidelines were presented to methodologically support a practical mapping.

Concluding, the first two research questions were answered by chapters 3 and 4. The third research question was answered using the matches found in chapters 3 and 4. The differences lead to a number of guidelines for BiZZdesigner modelling in chapter 5, which covers the methodological support (fourth research question) of the proposed mapping. All in all, the mapping from BiZZdesigner to Mendix microflows is less trivial than was expected at the start of this research.

Note that even though the process modeller already has to adjust his BiZZdesigner models to fit the guidelines, this mapping does not lead to directly executable models in Mendix, it will still require additional input. For example, Mendix microflows can require inputobjects, local variables, etcetera.

6.2 Relevance

Relevance of this thesis is divided into two categories, being practical and scientific. The practical relevance is threefold. To start with, the research offers insights for BiZZdesign and Mendix into the capabilities of their modelling languages due to the ontological and control flow analysis that were performed. Secondly, these results were compared to each other and used to create an overview of mappable and non-mappable constructs and behaviour between the two languages. Third, chapter 5 contains methodological support for this mapping from BiZZdesigner to Mendix. It contains practical guidelines and an example process that can be used when using a BiZZdesigner model as a start for Mendix system modelling.

Scientific relevance of this research lies in the combination of the business process modelling (BPM) and model driven software development (MDSD) worlds. This research also contributes on the literature on Bunge-Wand-Weber and Control Flow pattern analyses by applying them to two languages that have never been analyzed like this before. Relevance is also found for articles like [RM06] and [AAL07] that suggest using both analyses as a framework for mapping two languages to each other.

6.3 Limitations

As every research has its limits, so does this thesis. For starters, there is more than just one possible analysis for ontology and behaviour. Even though the ones used are by far the most common and acknowledged, it is possible that other analysis could show other issues. Another limitation is the use of control-flow patterns only, rather than also including a resource and data pattern analysis. Also, up to this date, there has been only one process which was worked out starting from a BiZZdesigner model, which was presented in section 5.4. More cases may lead to more specific guidelines when modelling in BiZZdesigner for a mapping to Mendix.

6.4 Future research and recommendations

The first recommendation and future research point is to find out if it is worth the effort to create an automated model transformation directly to Mendix. Considering the number of mismatches, it is recommended that other model execution platforms are also explored before making this decision. If the mapping would be realized, research should be done on how to make sure the guidelines are used and possibly even automatically performed. For example, creating additional actions for the item operations should be possible when transforming a BiZZdesigner model to Mendix.

In the future, it would also be interesting to be able to not only exchange models from BiZZdesigner to Mendix, but also in the other direction. This would be interesting as updating the model by hand can be error-prone as well as costly, which may result in “shelf-ware” BiZZdesigner models as they do not get updated properly. To do so, exploring QVT [OMG05] more would be a proper course of action.

More future research could be directed into finding out the best way to represent links between the originating business process and extended models in Mendix. This includes how to properly maintain those links when either the business process model or the implementing items get changed around.

Another interesting point for future research is use of information gathered by the implemented system, and then to (automatically) update the business process model accordingly. For example, if there is a OR-split present which has a 60/40 chance division in BiZZdesigner, but practice shows that the real division should be 50/50, this could be very valuable information for the various analysis that BiZZdesigner can perform.

6.5 Current implementation and recommendations

In parallel to this research, an export was created to have processes from BiZZdesigner be mapped to Mendix, but not directly to microflows. This was done using an XPDL export in BiZZdesigner, while having a new view called “Process View” in Mendix to support the import. An explanation including an example is provided in Appendix A for more information.

The first recommendation about this implementation is to check if it is possible to use the exported XPDL files in a more appropriate fashion in terms of execution. As Mendix is limited by their lack of parallelism, it is a good idea to find execution engines for XPDL/BPMN that may have a better match with the findings on BiZZdesigner's capabilities in chapter 3 and 4. This could also allow for more general conclusions about mapping a business process model to (MDA) software models.

If not, then it is recommended to perform research to look into what the exact use of the intermediate process explorer is for BiZZdesign compared to modelling business processes in that process explorer, without the use of BiZZdesigner.

Lastly, the developed Mendix process explorer has quite some improvements that can be made, such as icons and / or colours added to the gateways and support for representation of Items by using associations. Besides improvements like these, the model is unfortunately not able to execute such as microflows.

Appendix A: Current implementation process example

In parallel to this research, an export was created to have processes from BiZZdesigner be mapped to Mendix. This was done using an XPDL export in BiZZdesigner, while having a new view called “Process View” in Mendix to support the import.

Mendix process explorer

Due to the previously described issues with granularity and modelling level, Mendix has developed a new domain. This domain is the process explorer, which allows for more flexible process specification. The models in the process explorer are not executable and offer new functionalities that are more in line with the capabilities of BiZZdesigner. The process representation differs from the usual Mendix representation which was discussed in section 2.4. The process domain allows for the following constructs:

- Start event, End event, Sequence flow, Association flow, Annotation, all of which were described in section 2.4. The process domain supports multiple start events.
- Swim lane, representing the involvement of an actor in an activity.
- Activity, which can now have a sub process. In that case a “+” appears on the activity.
- Gateways, which can now be of the following (standard BPMN) types: Exclusive, Inclusive, Parallel and Complex. At this moment, the gateways are visually indistinguishable if you do not know their type.

Every event, activity and gateway can have “implementing items”, being the microflows, forms or rules which are used to support that particular construct. An overview of the concepts is given in Table 34.


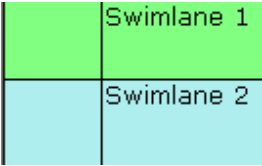

Representation	Name	Explanation
	Activity	The activity, in this case with corresponding sub process represents an action that needs to be performed. If there is no sub process connected to it, the “+” is not present.
	Swim lane	The swim lane represents an actor that needs to participate in the activities that are in that lane. Every swim lane can be given a colour to emphasize the different actors.
	Gateway	The gateway in the process explorer is now an all-in-one gateway, which can represent various types of splits and joins. The possible types are exclusive, inclusive, parallel and complex.

Table 34: Representation of new concepts in the Mendix process explorer

Advantages of the process domain are numerous for a mapping. A number of issues which were described previously are no longer a problem if the process explorer is used. A mapping is now possible for the following BiZZdesigner constructs:

- Actors
- And-split and And-join
- Blocks
- Implicit endpoints
- Multiple triggers

As for information losses, the process explorer now allows for representation of actors. The other losses remain present. An important note is that this new process view is non-executable, merely a BPMN style view on the XPDŁ export file.

The process is a complaint registration system of an editorial company, which accounts for the registration of complaints and handling compensation where needed. Every complaint is stored in a file, with the addition of compensation proposals if the complaint was accepted and properly specified. This section will start with the BiZZdesigner process model, followed by an initial mapping to Mendix without the new process explorer. After this, a mapping to the Mendix process explorer model will be shown.

The BiZZdesigner process model

To start with, a BiZZdesigner process model was created of this process. For readability in this thesis, the process was split into two pieces, which should not be seen as individual processes. In Figure 66, the first part of the process contains registration and assessment of the media office's complaint by the intermediary. If the complaint is accepted, a check for the cause of the complaint is done. If the cause is not specified (properly), this must be done so by the publisher.

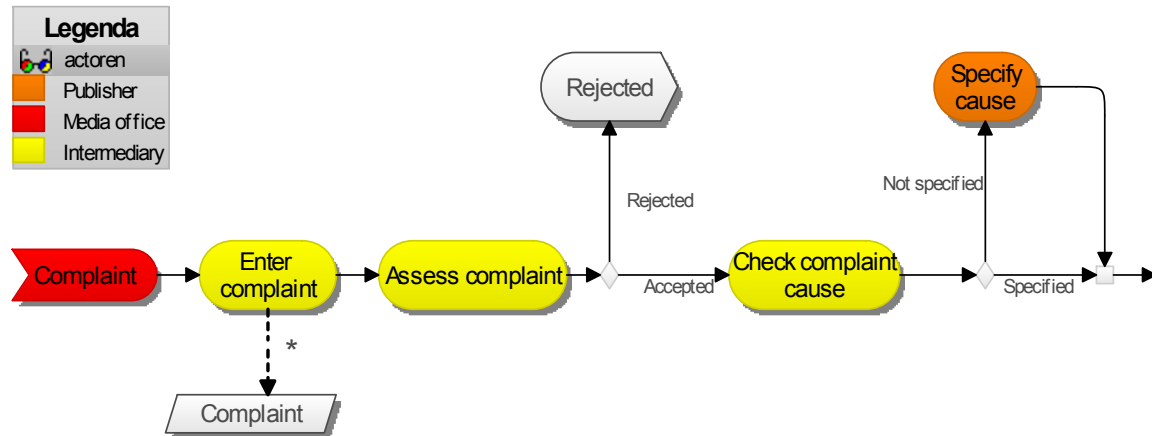


Figure 66: Part 1 of the complaint registration system in BiZZdesigner

After the cause of the complaint is determined, a compensation proposal can be created by the intermediary. This proposal then needs to be checked by the publisher, followed by acceptance or creation of a new compensation proposal. Accepted proposals then get assessed by the media office which submitted the complaint, which can result in a loop back to the creation of proposals or acceptance. When the proposal is accepted, the complaint and the proposal are archived and the process is finished.

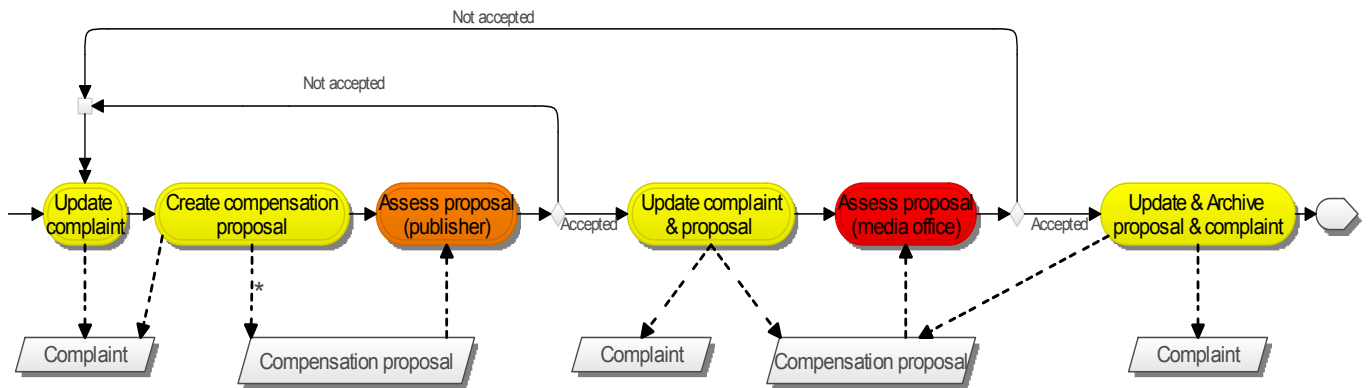


Figure 67: Part 2 of the registration process in BiZZdesigner

Mapping to a Mendix microflow

Coming from the business process model, a direct mapping to the Mendix microflows can be created. This mapping is shown in Figure 68 and Figure 69. The presented model has relatively few issues while mapping, considering the lack of problematic constructs like the AND-split and join, only a single trigger and no implicit termination in the example model. However, when taking a closer look, items cannot be represented in Mendix and the lack of an actor concept means it is unclear what actor is assessing the compensation proposal.

The solution for the item problem is to implement the item operation manually in an action that performs the correct CRUD operation on the specified item, which for Mendix should be stored in the datatype domain as well. For example, this means that the create complaint action becomes an action of the “object, create” type, and the object created is of the “complaint” datatype.

The lack of actor specification cannot be solved by the microflow of Mendix, therefore the choice was made to include the proper actor for the proposal assessments in the activity name to prevent confusion about who should be assessing.

Another problem here is the labels on the outgoing split arches, which are not present in Mendix. Although there is a similar looking labelling, this requires the split condition to be explicitly set, either by a variable, expression or rule.

In the provided example, the appropriate types have also been assigned to the mapped activities. However, many of these activities need a lot more refinement in order to become executable. An example is the “create compensation proposal” action. Creation of a compensation proposal could require a complaint cause, leading to addition of a check of the complaint data as well as a form

which the appropriate employee has to fill in. This form of course needs to be specified as well, including decisions like what fields are required, how the form is represented, etcetera.

It is clear that many activities are too abstract to be directly executable. This means that either the BiZZdesigner model needs to be revised to include more detail, or to accept manual refinement at the Mendix side, which illustrates the problems described in section 0.

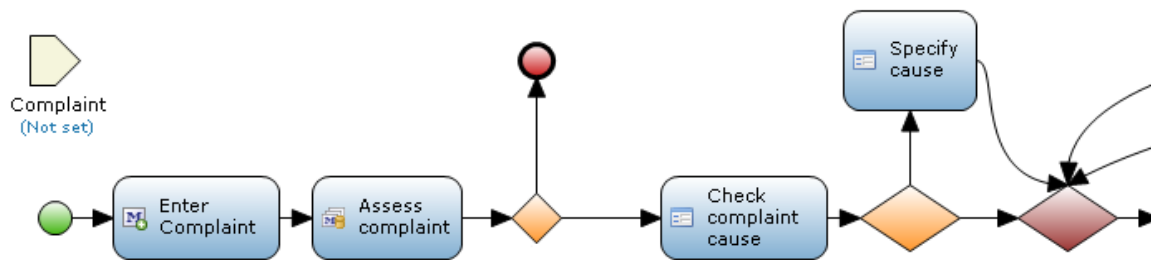


Figure 68: Part 1 of the registration process in microflow

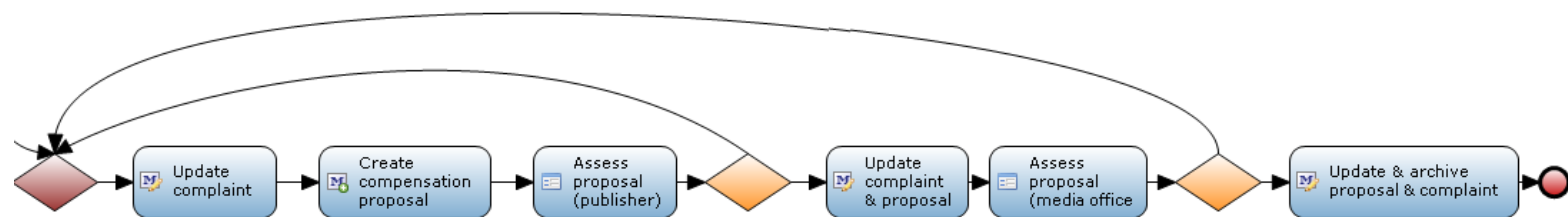


Figure 69: Part 2 of the registration process in microflow

Mapping to a Mendix process model

Using the newly developed BiZZdesigner export function, the BiZZdesigner process model is exported to XPD format. This file gets imported by the Mendix Business Modeller, which then gives a representation of the source model in their process explorer (BPMN format). The model then is ready for editing towards a working software solution.

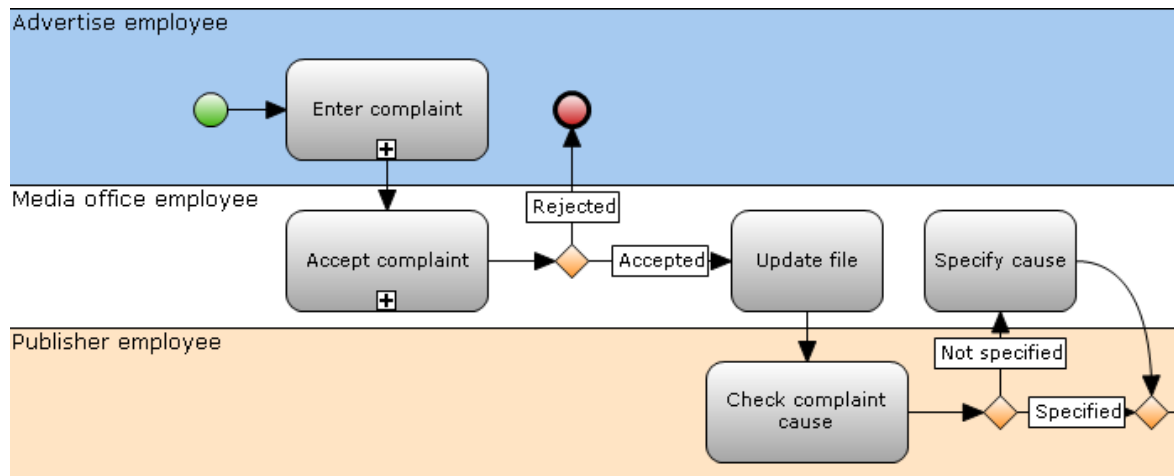


Figure 70: Part 1 of the registration process in the Mendix process explorer

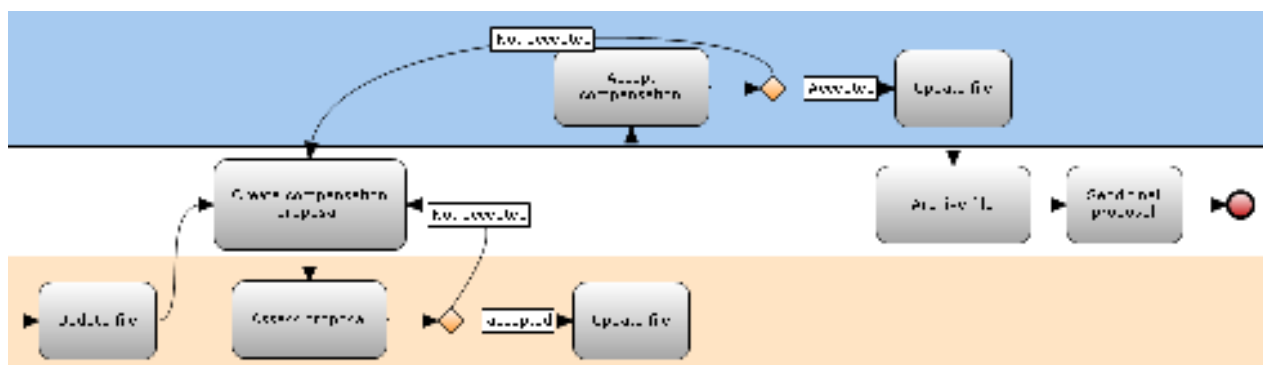


Figure 71: Part 2 of the registration process in the Mendix process explorer

The recently developed 2.4 version of Mendix supports import of XPD files. The import shows the imported process in the newly developed process explorer domain, showing the process in a lane-view ordered by performing actors. The process that is created by the import has extendable actions (called processes), which operate much like a block in BiZZdesigner. Individual actions can be coupled to implementing items which can be a microflow, but also things like forms. This process is not directly executable, but allows for analysts to see where implementation pieces are used.

To get to an executable model, a lot of detail needs to be added. This includes more information about the datatypes in Mendix, as well as microflows and specific forms for the user interface. However, first of all it is needed to create more specific activities to perform by both the system as well as humans. These activities are stored by adding new processes to the main process, in this case being “ComplaintRegistration”. In this particular example, “Enter Complaint” and “Assess Complaint” have a sub process with its own activities and implementing items.

As mentioned, the “Enter Complaint” activity translates to a number of steps (or sub-activities) for the employee. In this case, the employee needs to log in, navigate to the customer overview and create a new complaint file for that customer (see Figure 72). The description of such a process activity is getting very close to working instructions. Every process can have multiple implementing items, which in this example all have one connected to the steps. From left to right, these are a start page form for the login, a form showing the overview of customers from a media office perspective and a form to create a new complaint.

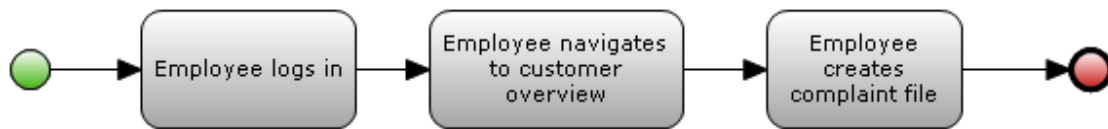


Figure 72: Enter complaint sub-activities in Mendix

Appendix B: Control flow for BiZZdesigner, XPDL and Mendix

Pattern		BiZZdesigner	XPDL	Mendix
1	Sequence	+	+	+
2	Parallel split	+	+	-
3	Synchronization	+	+	-
4	Exclusive choice	+	+	+
5	Simple merge	+	+	+
6	Multi-choice	+	+	-
7	Structured synchronizing merge	-	+	-
8	Multi-merge	-	+	-
9	Structured discriminator	+	+/-	+
10	Arbitrary cycles	+	+	+
11	Implicit termination	+	+	-
12	Multiple instances without synchronization	+	+	-
13	Multiple instances with <i>a priori</i> Design-Time knowledge	+	+	-
14	Multiple instances with <i>a priori</i> Run-Time knowledge	-	+	-
15	Multiple instances without <i>a priori</i> run-time knowledge	+	-	-
16	Deferred choice	+	+	-
17	Interleaved parallel routing	-	-	-
18	Milestone	-	-	-
19	Cancel activity	+	+	-
20	Cancel case	-	+	-

Table 35: Pattern support – XPDL taken from [RHAM06] (XPDL 2.0)

Pattern	BiZZdesigner	XPDL	Mendix
21 Structured loop	+	+	+
22 Recursion	-	-	+
23 Transient trigger	-	-	-
24 Persistent trigger	+	+	-
25 Cancel region	-	+/-	-
26 Cancel multiple instance activity	-	+	-
27 Complete multiple instance activity	-	-	-
28 Blocking discriminator	-	+/-	-
29 Cancelling discriminator	-	+	-
30 Structured partial join	-	+/-	-
31 Blocking partial join	-	+/-	-
32 Cancelling partial join	-	+/-	-
33 Generalized AND-join	-	+	-
34 Static partial join for multiple instances	-	+/-	-
35 Cancelling partial join for multiple instances	-	+/-	-
36 Dynamic partial join for multiple instances	-	-	-
37 Local synchronizing merge	-	-	-
38 General synchronizing merge	-	-	-
39 Critical section	-	-	-
40 Interleaved routing	-	+/-	-
41 Thread merge	+	+	-
42 Thread split	+	+	-
43 Explicit termination	-	+	+

Table 36: Pattern support – XPDL taken from [RHAM06] (XPDL 2.0)

References

- [AAL07] Aalst, W.M.P. van der, *Workflow patronen: een gereedschap voor het evalueren van BPM software*, Business Process Magazine, 13(4), pages 22-27, 2007.
- [AHKB03] van der Aalst, W.M.P., Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P., *Workflow Patterns*, Distributed and Parallel Databases, 14(3), pages 5-51, July 2003
- [BAU08] Bauler, P., Feltz, F., Frogneux, E., Rewart, B., Thomase, C., *Usage of Model Driven Engineering in the context of Business Process Management*, Paper for Multikonferenz Wirtschaftsinformatik 2008, pages 1963-1974, University of Belvaux, Luxembourg
- [BRO04] Brown, A.W., Model driven architecture: Principles and practice, Software and Systems modelling, Volume 3, Number 4, December 2004, p 314-327
- [BIZ03] BiZZdesign B.V., Handleiding BiZZdesigner, Enschede, 2003.
- [BIZ08] BiZZdesign 2008, *BiZZdesign website – History of BiZZdesign*. Retrieved at 16-05-2008 from <http://www.bizzdesign.nl/joomla/company/history.html>
- [BUN77] Bunge, M., Treatise on Basic Philosophy: Volume 3: Ontology I: The Furniture of the World. Reidel, Boston, 1977
- [BUN79] Bunge, M., Treatise on Basic Philosophy: Volume 4: Ontology II: A World of Systems. Reidel, Boston, 1979
- [DA04] Dehnert, J., Aalst, W. M. P. van der, *Bridging the gap between business models and workflow specifications*, International Journal of Cooperative Information Systems, Vol. 13, Issue 3, 2004, pages 289-332.
- [DGMR05] Davies, I., Green, P., Milton, S., Rosemann, M., *Analyzing and Comparing Ontologies with Meta-Models*, In J. Krogstie, T. Halpin and K. Siau (Ed.), Information Modeling Methods and Methodologies, Hershey, PA, USA: Idea Group, Pages: 1-16.
- [ECL09] Eclipse M2M website, <http://www.eclipse.org/m2m/>, retrieved at 15-02-2009.

- [EER99] Eertink, H., Janssen, W., Oude Luttighuis, P., Teeuw, W., Vissers, C., A Business Process Design Language, Lecture notes in Computer Science volume 1708, 1999, pages 76-95.
- [FLU06] Fluegge, M., Garcia dos Santos, I.J., Paivo Tizzo, N., Madeira, E.R.M., *Challenges and Techniques on the Road to Dynamically Compose Webservices*, ICWE06, Palo Alto, California, USA, 11-14 July 2006.
- [GAR03] Gardner, T., Griffin, C., Koehler, J., Hauser, R., *A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard*, IBM, July 2003.
- [GE07] Gehlert, A., Esswein, W., *Toward a formal research framework for ontological analyses*, Advanced Engineering Informatics, Volume 21, Issue 2, April 2007, Pages: 119-131.
- [GPS05] Guizzardi, G., Ferreira Pires, L., van Sinderen, M., *An Ontology-Based Approach for Evaluating the Domain Appropriateness and Comprehensibility Appropriateness of Modelling Languages*, Lecture Notes in Computer Science, Volume 3713/2005, November 2005, Pages: 691-705.
- [GR00] Green, P., Rosemann, M., *Integrated Process Modelling: An Ontological Evaluation*, Information Systems Vol. 25, No. 2, 2000, pages 73-87.
- [GRI05] Green, P., Rosemann, M., Indulska, M., *Ontological Evaluation of Enterprise Systems Interoperability Using ebXML*, IEEE Transactions on knowledge and data engineering, Volume 17, No. 5, May 2005, Pages: 713-725.
- [GRU93] Gruber, T., *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*, International Workshop on Formal Ontology, March 1993, Padova, Italy. <http://tomgruber.org/writing/onto-design.pdf>
- [GRU08] Gruber, T., to appear in *Encyclopedia of Database Systems*, Liu, L., Ozsü, T., Springer-Verslag 2008. <http://tomgruber.org/writing/ontology-definition-2007.htm> 30/1/09
- [GRE07] Green, P., Rosemann, M., Indulska, M., Manning, C., Candidate interoperability standards: An ontological overlap analysis, Data & Knowledge Engineering 62, 2007, pages 274-291.
- [HEV05] Havey, M., *Essential Business Process Modelling*, O'Reilly Media, August 2005.
- [HOO06a] Hoogeveen, S.C.T., B2B: BiZZdesigner to BPEL, The transformation of AMBER models to standard compliant process definitions for business process execution, Master Thesis University of Twente, February 2006
- [HOO06b] Hoogeveen, S.C.T., *Handboek BPEL versie 1.0*, BiZZdesign B.V., Enschede, July 2006.
- [JSL07] Jonkers, H., Steen, M.W.A., Heerink, L., van Leeuwen, D., *Bridging BPM and MDE: On the Integration of BiZZdesigner and OptimalJ*, Selected position paper for Eclipse Modeling Symposium, Eclipse Summit Europe 2007, October 9, 2007, Ludwigsburg, Germany.
- [KEN02] Kent, S., *Model Driven Engineering*, IFM 2002, LNCS 2335, 2002, pages: 286-298.

- [KWB03] Kleppe, A.G., Warmer, J.B., Bast, W., *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley, 2003.
- [MB02] Meller, S.J., Balcer, M.J., *Executable UML: A foundation for Model-Driven Architecture*, Addison-Wesley, 2002.
- [MED09] Medini QVT website, <http://projects.ikv.de/qvt>, retrieved at 15-02-2009.
- [MEN08] Mendix website 2008, *The Mendix Business Improvement Architecture*. Retrieved at 20-05-2008 from http://www.mendix.com/site/?page_id=4
- [MRI07] Meuhlen, M. zur, Recker, J., Indulska, M., *Sometimes Less is More: Are Process Modeling Languages Overly Complex?*, Eleventh International IEEE EDOC conference workshop, 2007.
- [MUE04] Muehlen, M. zur, *Workflow-based Process Controlling. Foundation, Design and Application of workflow-driven Process Information Systems*. Logos, Berlin 2004.
- [MXDN08] Mendix Developer Network, *Wiki Mendix Knowledge base*, retrieved at 29-06-08 from <https://projects.mendix.nl/trac/KnowledgeBase/wiki/>
- [MXT08] Mendix 2.3 training course, *Micro-flow, nesting and business rules*, 2008.
- [OH02] Opdahl, A. L., Henderson-Sellers, B., *Ontological Evaluation of the UML Using the Bunge-Wand-Weber Model*, *Software and Systems Modelling*, 1(1), 2002, pages 43-67.
- [OMG00] Object Management Group, Whitepaper: *Model Driven Architecture Draft 3.2*, November 2000, retrieved at 14-02-2009 from <http://www.omg.org/docs/omg/00-11-05.pdf>
- [OMG01] Object Management Group, *Model Driven Architecture (MDA)*, Architecture Board ORMSC, June 2001, retrieved at 14-02-2009 from <http://www.omg.org/docs/ormsc/01-06-01.pdf>
- [OMG03] Object Management Group, *MDA Guide version 1.0.1*. Retrieved at 16-05-2008 from <http://www.omg.org/docs/omg/03-06-01.pdf>
- [OMG05] Object Management Group, *Meta Object Facility (MOF) 2.0 Query/View/Transformation specification*, adopted final specification, November 2005. Retrieved at 23-01-2009 from <http://www.omg.org/docs/ptc/05-11-01.pdf>
- [OMG08] Object Management Group, *MDA success stories*. Retrieved at 16-05-2008 from http://www.omg.org/mda/products_success.htm
- [PRP06] Perez, J.M., Ruiz, F., Piattini, M., *MDE for BPM, A Systemic Review*, ICISOFT September 11-14th 2006, in *Communications in Computer and Information Science* volume 10, p127-135, July 2008.
- [RFP07] Rodriguez, A., Fernandez-Medina, E., Piattini, M., 2007, in *IFIP International Federation for Information Processing*, Volume 255, *Research and Practical Issues of Enterprise Information Systems II Volume 2*, eds. L. Xu, Tjoa A., Chaudhry S. (Boston: Springer), pp. 1239-1249.
- [RHAM06] Russel, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N., *Workflow Control Patterns: A revised view*, BPM Center Report BPM-06-22, 2006.

- [RHA06] Russel, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., *Exception Handling Patterns in Process-Aware Information Systems*, BPM Center Report BPM-06-04, BPMcenter.org, 2006.
- [RHEA04a] Russel, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P., *Workflow Data Patterns*, QUT Technical report, Queensland University of Technology, Brisbane, 2004.
- [RHEA04b] Russel, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P., *Workflow Resource Patterns*, BETA Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven, 2004.
- [RIRG05] Recker, J., Indulska, M., Rosemann, M., Green, P., *Do Process Modelling Techniques Get Better? A Comparative Ontological Analysis of BPMN*, Proceedings of the 16th Australasian Conference on Information Systems (ACIS). Australasian Chapter of the Association for Information Systems, Sydney, Australia, December 2005.
- [RM06] Recker, J., Mendling, J., *On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modelling Languages*, CAISE 2006 Workshop proceedings – Eleventh International Workshop on Exploring Modelling Methods in System Analysis and Design, Luxembourg, 2006.
- [RRIG06] Rosemann, M., Recker, J., Indulska, M., Green, P., *A Study of the Evolution of the Representational Capabilities of Process Modelling Grammars*, Lecture Notes in Computer Science, Volume 4001/2006, July 2006, Pages: 447-461.
- [RWR06] Recker, J., Wohed, P., Rosemann, M., *Representation Theory Versus Workflow Patterns – The Case of BPMN*, Lecture notes in computer science, October 2006, pages 68-83.
- [SCH06] Schmidt, D.C., *Guest Editor's Introduction: Model Driven Engineering*, Computer, vol. 39, no. 2, Feb. 2006, pages: 25-31.
- [SCF97] Suleiman, M., Cart, M., Ferrié, J., *Serialization of concurrent operations in a distributed collaborative environment*, Proceedings of the international ACM SIGGROUP conference on Supporting group work: the integration challenge, 1997.
- [SK03] Sendall, S., Kozaczynski, W., *Model Transformation: the Heart and Soul of Model Driven Software Development*, EEE Software, vol. 20, no. 5, Sep./Oct. 2003, pages: 42-45
- [WIE08] Wieringa, R., *Research and Design Methodology for Software and Information Engineers*, University of Twente, Netherlands, January 3rd, 2008.
- [WIL77] Williams, M.H., *Generating structure workflow diagrams: the nature of unstructuredness*, Computer Journal 20 (1), 1977, pages: 45-50.
- [WFMC08] Workflow Management Coalition, *Process Definition Interface – XML Process Definition Language version 2.1* (WFMC-TC-1025), Final approved version, March 24th 2008.
- [WFP08] Control-Flow patterns, *Workflow patterns website – Patterns*. Retrieved at 23-06-2008 from <http://www.workflowpatterns.com/patterns/control/index.php>

- [WW90] Wand, Y., Weber, R., *An ontological model of an information system*, IEEE Transactions on Software Engineering Volume 16 , Issue 11 (November 1990), Pages: 1282 - 1292.
- [WW06] Wand, Y., Weber, R., *On Ontological Foundations of Conceptual Modeling: A response to Wyssusek*, Scandinavian Journal of Information Systems, 2006, 18(1), Pages: 127-138.
- [WYS06] Wyssusek, B., *On Ontological Foundation of Conceptual Modeling*, Scandinavian Journal of Information Systems, 2006, 18(1), Pages: 63-80.