Application of Model Transformations and State Spaces for Verification of Diagrammatic Workflow Models

Author: Endre Vestbø
eve034 / h119265

Supervisors: Dr. Lars Kristensen
Dr. Yngve Lamo

September 1, 2014
Abstract

Master in Software Engineering

Application of Model Transformation and State Spaces for Verification of Diagrammatic Workflow Models

Behavioral verification is an important aspect in software engineering. Lack of proper software testing can lead to high expenses and in worst case scenarios to serious accidents. This has lead to a boom in tools for automatic testing of source code. State space exploration is a key concept for behavioral verification in Model Driven Engineering (MDE).

This thesis investigates how we can describe the semantics of DERF, a diagrammatic workflow modeling language, by means of model transformations for state space exploration. A state space is a set of states where states are connected by directed edges, often called transitions. A transition from one state to another usually represents some behavior taking place and thereby changing the state of the system. State space exploration is the term commonly used for generating a state space. The method used to explore the state space is through the application of model transformations. For this we will interpret the semantic meaning of DERF models. DERFs semantics are implemented through metamodeling and diagrammatic constraints and can almost be directly translated into model transformation rules. These transformations rules are then used to generating the states of the state space. By predefining the transformation rules based on the semantics of DERF, the state space exploration can be automated for any DERF models. But generating state spaces has proven to create problems when it comes to memory consumption. The basic method for state space exploration, which will be investigated, keeps all the states from the state space in memory. An approach called the sweep-line method will be examined to alleviate this problem. The sweep-line method uses information
about progress in the state space to limit the number of states held in memory by deleting obsolete states. An obsolete state is a state that is no longer required to explore the rest of the state space.
Acknowledgments

First and foremost I want to express my gratitude to my supervisors, Dr. Lars Kristensen and Dr. Yngve Lamo. They have helped me with useful remarks and comments in addition to have shown remarkable patience. Further, I’d like to thank my friend and college Petter Barvik who has helped me on the way with tips about the tools used in this thesis and for sharing his insight on model transformations. I’d also like to thank PhD candidate Xiaoliang Wang who has quickly fixed bugs in the DPF Workbench when they put a halt in my implementation. Lastly, I’d like to thank my friends and family who has helped me through the process of thesis writing by giving me moral support. I am grateful for having the pleasure to know and work with you all.
Contents

1 Introduction .............................................. 1
   1.1 Thesis Aim and Results ............................. 5
   1.2 Thesis Outline ...................................... 7

2 DERF - A Workflow Modeling Language ................. 8
   2.1 Workflow Modeling .................................. 8
   2.2 Modeling Languages ................................. 9
   2.3 The Diagrammatic Predicate Framework ............ 12
      2.3.1 Metamodelling .................................. 12
      2.3.2 Constraint Modelling .......................... 16
      2.3.3 DPF Summary .................................. 18
   2.4 The DERF Workflow Language ....................... 19
      2.4.1 DERF - Metamodelling ......................... 19
      2.4.2 DERF - Constraint Modelling ................. 22

3 Model Transformation ...................................... 29
   3.1 Model Transformations in General ................. 29
   3.2 Transformation Rules for DERF ..................... 31

4 State Space Exploration and Model Verification ........ 39
   4.1 State Space Exploration ............................ 39
   4.2 Automatic State Space Exploration for DERF .... 40
   4.3 Verifying Model Behavior ......................... 46
   4.4 State Space Explosion .............................. 48
   4.5 The Sweep-line Method ............................. 49
      4.5.1 Verifying properties with the sweep-line method .... 53
5 Implementation

5.1 Overview

5.2 Implementing the DERF Language

5.2.1 EMF and ECore

5.2.2 DPF Workbench

5.2.3 Implementing DERF in the DPF Workbench

5.3 Implementing the Model Transformations

5.3.1 The Henshin Project

5.3.2 Defining transformation rules

5.3.3 Transforming Models

5.3.4 Using Henshin to transform DERF models

5.3.5 The [NodeMult, n] predicate

5.4 Automatic State space Exploring Using Henshin

5.4.1 Approach A

5.4.2 Approach B

5.5 Implementing the Sweep-line Algorithm

5.5.1 The sweep-line algorithm

5.5.2 Progress measures

6 Evaluation and Results

6.1 Using the prototype

6.2 Test models

6.2.1 Model A

6.2.2 Model B

6.2.3 Model C

6.2.4 Model D

6.3 The Normal State Space Exploration Method

6.3.1 Comparing Approach A and Approach B

6.4 The Sweep-Line Method

6.4.1 Progress measure strategy results
7 Conclusions and Future Work

7.1 Conclusions ................................................................. 88
7.2 Future Work ............................................................... 89
    7.2.1 Improving transformation rule approach ..................... 89
    7.2.2 Create better progress measures ............................. 89
    7.2.3 Verifying Model Behavior .................................... 90
    7.2.4 Creating an integrated DERF tool .......................... 90

References ................................................................. 91

8 Attachments
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A UML Activity Diagram based on [34]</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Directed graph example</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Graph homomorphism example</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>DPF metamodeling example</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>A sample signature for DPF [32]</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>DPF example of application of the [irr] constraint</td>
<td>17</td>
</tr>
<tr>
<td>2.7</td>
<td>A specification $M_n$, its type $M_{n+1}$ together with a pullback of the</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>[irr] predicate</td>
<td></td>
</tr>
<tr>
<td>2.8</td>
<td>Metamodelling hierarchy in DPF [32]</td>
<td>19</td>
</tr>
<tr>
<td>2.9</td>
<td>A small hospital procedure represented in simplified DERF, based on [34]</td>
<td>20</td>
</tr>
<tr>
<td>2.10</td>
<td>The DERF meta hierarchy</td>
<td>22</td>
</tr>
<tr>
<td>2.11</td>
<td>DERF $M_4$ Predicates</td>
<td>23</td>
</tr>
<tr>
<td>2.12</td>
<td>DERF $M_1$ Predicates $[\text{and_split}]$ and $[\text{and_join}]$</td>
<td>24</td>
</tr>
<tr>
<td>2.13</td>
<td>DERF $M_5$ Predicates $[\text{xor_split}]$ and $[\text{xor_join}]$</td>
<td>25</td>
</tr>
<tr>
<td>2.14</td>
<td>Valid example extension to the semantic interpretation of $\text{xor_join}$</td>
<td>25</td>
</tr>
<tr>
<td>2.15</td>
<td>DERF $M_4$ Predicates $[\text{NodeMult}]$</td>
<td>26</td>
</tr>
<tr>
<td>2.16</td>
<td>Valid example extension to the semantic interpretation of $\text{NodeMult}$</td>
<td>26</td>
</tr>
<tr>
<td>2.17</td>
<td>A small hospital procedure represented in DERF. [34]</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Principle behind the double pushout approach [5]</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>Rule: Enabled to running</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>Rule: Simple Arrow</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Rule: $[\text{and_split}]$</td>
<td>35</td>
</tr>
</tbody>
</table>
3.6 A simplified version of all rules. .................................. 36
3.7 An example of transformation rules in use ...................... 37

4.1 A small hospital procedure represented in DERF. [34] .......... 42
4.2 Statespace states for Figure 4.1 .................................. 43
4.3 Partial state spaces for the model in Figure 4.1 ................... 44
4.4 Full statespace states for the model in Figure 4.1 ............... 46
4.5 Example sweep-line layers, statespace extracted from Figure 4.4 .......................................................... 50
4.6 An optimal sweep-line layer division ................................ 51
4.7 Statespace states for Figure 4.1 .................................. 55

5.1 Overview of experiment implementation .......................... 60
5.2 The visual EMF ECore editor showing a simple model created in ECore. [3] ......................................................... 62
5.3 The DPF Metamodel .................................................. 63
5.4 Metamodeling with the DPF Workbench [22] .................... 63
5.5 An extract of the DPFCore model showing graphs, nodes, arrows and typing ......................................................... 64
5.6 An extract of the DPFCore model showing constraints and predicates on Nodes ......................................................... 65
5.7 An extract of the DPFCore model showing constraints and predicates on Arrows ......................................................... 65
5.8 DERF implementation in DPF ....................................... 66
5.9 Using the DPF Workbench Signature Editor to implement DERF constraints ......................................................... 67
5.10 Partial implementation of the simple arrow transformation from Figure 3.4 in Henshin .................................................... 70
5.11 A partial implementation of the transformation rule specified in Figure 3.3 in Henshin .................................................... 72
5.12 A partial implementation of the transformation rule specified in Figure 3.3 in Henshin .................................................... 75
5.13 D predicate in DERF $M_0$ ........................................ 75
5.14 Alternative partial implementation of the simple arrow transformation from Figure 3.4 in Henshin .................................................... 76
8.1 Test model B
8.2 Test model C
8.3 Test model D
Chapter 1

Introduction

The prerequisite knowledge for this thesis is a bachelor in computer science / software engineering or equivalent. It is assumed that the reader has basic understanding of UML modeling or an equivalent modeling toolset, has some knowledge about software testing and is familiar with standard data structures such as queues and graphs.

The art of modeling has been around a long time and has many areas of application. From creating model structures of future constructions such as deep sea drilling platforms to specifying business processes in workflows. In terms of applications, the reasons of modeling are diverse. One example would be the need for simplification and abstraction from implementation details, for example platform specific aspects in software engineering. Another example is to represent complex natural systems that are near impossible to recreate in detail.

In software engineering, modeling is often used as a tool for design and documentation. There exists many different modeling languages for a variety of application areas. As an example, Entity Relationship [44] models are used to represent database schemes. UML [30] class diagrams represents the structure of code at a high abstraction level, while UML activity diagrams may represent the behavior of processes or software. Models can be divided into different types based on what they represents. UML class diagrams and entity-relationship diagrams are structural models. As the name implies, structural models represents structures.
UML activity diagrams can be used to create workflow models. A workflow model represents progression and are commonly represented as a directed graph consisting of tasks as nodes and directed arrows between them that represents the order in which these tasks can be completed. For example, on a car rental website, a user can first search for a free car, then he can place an order on the car and finally pick it up. Many software development paradigms, such as the Rapid Unified Process paradigm, encourage the use of modeling for software design, both structural and for business logic, and documentation.

In this thesis we will look at a workflow modeling language called DERF. The name DERF comes from the four different states a task can have, Disabled, Enabled, Running or Finished. DERF consists of tasks and flows between these tasks. Similar to UML activity diagrams, DERF also includes control structures. Also, both these modeling languages are able to specify that a set of tasks must be completed before a next one can start. One of the areas DERF differs from UML activity diagrams is that it has a sense of progress in the actual tasks themselves.

Another way to categorize modeling languages is to distinguish general purpose and domain specific languages. Domain specific modeling languages opens up the possibility of letting domain experts implement the business logic of software through modeling. For example, within health-care, doctors, nurses and other on-site staffs could be able to create and edit software systems by using a graphical editor with health-care specific elements. DERF is actually developed based on healthcare processes.

Modeling languages comes in many different flavours. Some, as the ones mentioned above, are graphical. Other, like eXtensible Markup Language, are textual. Model Driven Engineering (MDE) is a field where models are seen as the main artifact of software. This requires formalized modeling languages that can be machine interpreted. Ecore, a language supported by the Eclipse Modeling Framework (EMF) is an example of such a language. EMF allows users to use models as part of the software implementation through partial code generation. The term code generation means to generate valid software code from models. A partial code generation only generates parts of the code that is needed to obtain running software. For instance, EMF can be used to create code templates from class diagrams, but the user has to fill in the logic of the
software. Other languages, such as Coloured Petri Nets [19] (CPN) can be used to prototype behavior. Behavior can span from the order tasks are being done, like in workflow modeling, to more complex systems like network protocols. In MDE model transformations are used to take the models out of their statical shell and make them more useful. A model transformation is a method to formalize how a model can be transformed into something else. They are divided into two groups, model-to-text and model-to-model. EMF relies on model-to-text transformations to create code from a class diagrams. In this thesis we will look at another usage of model transformations. We will look at how model-to-model transformations can add behavior to DERF models. A model-to-model transformation takes a model as input and outputs a new one. For example, lets say we have a simple model with one task Write thesis that is labeled Running. We could then specify a transformation that takes that model as input and outputs a new model with the same task, Write thesis, only now it is Finished. This example illustrates how we can use model transformations to execute DERF models similar to how classic software runs by altering the state of the task.

One of the large challenges in software engineering are logical errors. Introduction of logical errors has grown with the size and complexity of modern software, as has the price of development. An case study from the US, more specifically the governmental National Institute of Standards and Technologies, NIST, published in 2002 estimates that faulty software costs the state 59.5 billion USD (2002) annually [26]. Another case study, entitled 'The Economic Impacts of Inadequant Infrastructure For Software Testing' [38], looked at the impact of inadequate software testing in the U.S. automotive and aerospace industry. The study is based on 10 software vendors and 179 end-users. Taking both the time spent on fixing bugs and the consequences of using this software, he estimated the cost to be over 1.8 billion US dollars (2002). While the potential cost reduction from ”feasible infrastructure improvements” [38] is estimated to be a little below 0.6 billion US dollars (2002). Many modern paradigms for traditional software engineering development, like eXtreme Programming, emphasizes on cohesive software testing.

In MDE, a method called state space exploration can be used to verify behavioral properties of models. As an example [19] shows how Ericsson Telebit project used
CPN to prototype and test the logic of their *Edge Router Discovery Protocol* (ERDP) through state space exploration. A state represents a property of a model. Like in our DERF example, a state can be the model itself with *Write thesis* labelled *Running*. A state space is a set of states and transitions between them. Exploring a state space means to generate all states that are reachable from some initial state. For DERF, a natural choice for a transition can go from [Write thesis Running] to [Write thesis Finished]. In itself, the state space is now a graph, with states as nodes that are DERF models and directed arrows that shows the progress of how we can go from one model state to the next. The actual transition is then the model transformation described above. To sum it up, we use model-to-model transformations to explore the state space. The states are DERF models and the transitions corresponds to model transformations.

State space exploration has a drawback when it comes to resource consumption. There exists many different techniques to limit resource consumption. One of these is called the sweep-line method [6 19 20] and can be used to minimize memory consumption. The sweep-line method relies on a notion of progress in the state space exploration and tries to eliminate explored states that are not needed for future exploration. Since DERF is a workflow modeling language it has a keen notion of progress. This thesis investigates how we can use the sweep-line method in state space exploration of DERF models.
1.1 Thesis Aim and Results

State space exploring is an important aspect in MDE and is used to verify behavioral properties of models. To exploring a models state space is to find progression that takes us from one state to another. The modeling language that is used in this thesis is called DERF. DERF is a graph based workflow language and consists of typical workflow semantics and elements. The elements are tasks and flows. The tasks are the node of the graph and the flows are directed arrows connecting the different tasks. Flows represent how we can go from one task to the other. The semantics of DERF includes combining tasks: A and B and splitting tasks: only A or only B. In addition, DERF also includes semantics that defines progression in the workflow model, allowing tasks to be either enabled, running or finished. This sense of progression can be used to define states for DERF. For a simple DERF model, with one task A, one state can be that A is enabled, while another can be that A is running.

This thesis explores how model transformations can be used to explore state spaces. [33, 34] defines formalized rules in a transition system for how DERF models can change states. These rules are based on the workflow structures for combining and splitting tasks as well as how tasks can go from enabled, to running and to finished. This leads to the first problem this thesis will look at:

- How can we translate the semantics of DERF into transformation rules that can be used for state space exploration?

There are two problems that must be solved to answer this question. First, DERF must be implemented. Because DERF is defined by the Diagrammatic Predicate Framework [7, 32] (DPF), The DPF Workbench [22] becomes a natural choice for implementing DERF, as it is an implementation of DPF. The DPF Workbench is implemented in the Eclipse Modelling Framework (EMF) and is build upon the modeling language ECore [10].

Further, we will have to look at how model transformations are defined. The Henshin project [39] is a natural choice for defining model transformations. It is created to define model transformations for EMF and provides all the transformation tools that is needed for this thesis. This includes both a graph based language for defining transformation rules and tools for preforming the model
transformations. To solve our problem we will translate the DERF semantics into Henshin transformation rules and use them to explore the state space.

A common problem in state space exploration is state space explosion. State space explosion is a term that is used when referring to the substantial number of states a state spaces can contain. State space explosion causes problem regarding to computer memory limitations. In this thesis we will look at a technique called the sweep-line method that can be used to reduce the number of states we need to keep in memory during the state space exploration. The sweep-line method exploits the notion of progress through the state space and tries to identify states that are no longer needed to reach the rest of the state space and delete them. This is done by using progress measures. A progress measure, as the name implies, is a measure of progress in the state space. Each state has a progress value and the sweep-line method uses these values to decide if the state can be deleted by exploring them in the least-order. The second problem this thesis will explore is therefore:

- How can we define progress measures for DERF to use with the sweep-line state space exploration method?

The goal of a progress measure is to minimize the amount of states held in memory. To solve this problem we will again look to the semantics of DERF. Given some input DERF model, we need a method to define progress measures that later can be used to calculate the progress values for states in the state space.

The results from this thesis provides methods for translating the DERF semantics into transformation tools, using them for state space exploring and definitions of how we can create progress measures for DERF.
1.2 Thesis Outline

Chapter 2
This chapter introduces workflow modeling in the MDE practice, the Diagrammatic Predicate Framework (DPF) and the DERF modeling language. DPF is a framework created around the idea of using metamodeling to create domain specific languages in a common environment. The DERF language will be the foundation when exploring how to use model transformations to generate a state space.

Chapter 3
In Chapter 3 we describe how model transformations can be used to interpret DERF models in an execution environment by translating DERFs semantics into transformation rules.

Chapter 4
This chapter explains state spaces, how they can be generated using model transformations, and how we can use them to verify behavioral properties of models. The chapter also explores how the DERF semantics can be used to generate automatic progress measures for memory efficient state space exploration using the sweep-line method.

Chapter 5
This chapter presents the tools and methods used to implement the techniques applied in the thesis. These includes the DPF Workbench, an implementation of DPF, and Henshin which is a diagrammatic model transformation tool.

Chapter 6
The results and evaluation chapter goes through the results from Chapter 5 and compare and evaluates them.

Chapter 7
This chapter sums up the results and experiences gained through this thesis and future work related to the problems visited in this thesis.
Chapter 2

DERF - A Workflow Modeling Language

This chapter explores how a workflow modeling language can be defined in a formal manner for application in MDE. First we will look at workflow modeling and modeling languages from a more general perspective. Next is the Diagrammatic Predicate Framework which has been used to specify the DERF language. Lastly we will take DERF apart and explore the semantics and syntax of the language.

2.1 Workflow Modeling

A workflow is a system of activities together with flows defining the order in which the activities can be executed. For example, a hospital workflow might consists of different procedures and protocols a patient must go through from filling out a form at the reception desk until his treatment is done. Another example is a production environment, where some entities goes through several phases of processing, such as molding, assembly and painting until it is shipped to customers. Such workflows are often visualized graphically with models. They provide an abstract overview and serves as both documentation of existing processes and as a planning tool for future processes. Because of their versatility and wide applicability, many different standards of workflow modeling languages has been created. This ranges from simple languages only containing activities and flows connecting them to more advanced concepts applying time and flow management such as exclusive-or
statements. One of the most widely used workflow modeling languages is the Business Process Model and Notation [29] (BPMN) language. The goal of BPMN is to be understandable for all people involved in creating and maintaining the models. Another popular workflow language within software engineering is the activity diagrams of the Unified Modeling Language [30] (UML). UML is a general purpose modeling language created for software engineering. It is developed in the context of the Object Management Group [28] (OMG) and was approved as an ISO standard in 2000. UML consists of several sub-languages for different purposes, including both class diagrams and activity diagrams. Figure 2.1 shows an example of a UML activity diagram. It contains nodes, control structures and flows. Each node represents a specific activity. Figure 2.1 models a health-care process, where each activity involves a patient. The flows between these activities are marked with directed arrows. The control structures includes a starting point (a solid black circle) and an end point (a circle with a black center and a black line around it). Further there are some solid bars. If we imagine running this model as a computer program, the solid bars means that all flows (and their accompany incoming activities) must have been executed when going from the bar. When leaving a bar, all ingoing flows (and the activities leading to these flows) should be processed. So after InitialEvaluation both Bloodtest1 and MRI follow. And before Evaluation1, both Bloodtest1 and MRI must have been concluded. The last control structure in Figure 2.1 is the diamond shape which represents an exclusive-or statement. That is, after Evaluation1 either ProcedureA or ProcedureB can follow, but not both. This model only utilizes a subset of the UML activity diagram language. It is also possible to add user-defined text to the flows, often used to define how a decision is made on the exclusive-or diamonds. The UML activity diagrams covers the basics of workflow modeling by defining an activity as a process and a flow between them. UML activity diagrams are good for informal use by humans, but not very usable for machines. In MDE, strict formalized modeling languages must be defined to make them suitable for machine execution.

2.2 Modeling Languages

In [5], models platforms are divided into three layers. Computation-Independent Model (CIM), Platform-Independent Model (PIM) and Platform-Specific Model
Figure 2.1: A UML Activity Diagram based on [34]
Computation-Independent models are regarded as the most abstract level. Belonging to this category we find models that has no IT-related elements and remain independent of whether and how a system is implemented. They are often referred to as business models because they use syntax that is based on a specific domain \[5\]. An example of a Platform-Independent model is the *Enhanced Entity Relationship* (EER) model. EER is used to model database schemes, but it does not cover platform specifics such as which datatypes the entries are.

A Platform-Specific Model must contain all the information needed for behavior and structure in regards to the platform in question. An example of a platform could the database management system MySQL or maybe the Pentium II processor. Here, the EER modeling language does not suffice because it lacks syntax for important aspects such as data types \[5\].

So far the modelling languages EER and UML activity diagrams has been mentioned. In MDE, modelling languages are a lot like classic programming languages in software engineering. They are a group of languages designed to be represented and interpreted by a computer. They have semantic and syntactical rules that must be obeyed for the model to be valid and compilable. In addition, modelling languages have both abstract and concrete syntax. The concrete syntax represents the building blocks developers use to construct models. The concrete syntax is then parsed into abstract syntax so it can be validated based on its semantic rules and is represented in a suitable manner depending on purpose.

As in classical modeling languages, MDE languages comes in different kinds of flavors. Being either textual like XML, graphical like UML or a mix of both. Structural languages like UML Class Diagrams describes mainly system structures while behavioral languages like UML Activity Diagrams describes a systems behavior. In fact, a workflow model is a description of behavior in the form of a process. Therefore, a workflow modeling language is a behavioral modeling language.

Further, there exists both general-purpose, and domain specific languages. *Domain Specific Modeling Languages* (DSML) is designed to solve problems in specific domains, such as physics simulation or health-care. *General Purpose Modeling Languages* (GPML) are meant to cover all problem domains. An example
of a GPML language is the UML activity diagram. Even though it has a domain, workflow modelling, it can be used to create all kinds of workflow models.

2.3 The Diagrammatic Predicate Framework

The Diagrammatic Predicate Framework (DPF) is an ongoing research project initiated by Bergen University College and the University in Bergen in 2006. The framework comes complete with a tool set called the DPF Workbench. The goal of DPF is to create a general purpose diagrammatic modeling framework that is formalized in category theory and to be used to create domain specific modeling languages. A diagram is a graphical symbolic representation of information. There are two important concepts behind DPF: metamodeling and model constraints. DPF and its concepts plays a vital role in how the DERF language is defined.

2.3.1 Metamodeling

MDE model languages has both an abstract and a concrete syntax, much like classical programming languages such as Java and C. The abstract syntax defines all valid models a language can implement. In MDE, the abstract syntax is often defined by a model. This is done by a technique called metamodeling. A metamodel defines what elements are available and how they can be related. OMGs Meta-Object Facilities MOF models depend upon such a structure with four layers. In meta-modeling, all objects on a given layer must have a relation to an element on the layer above, with the exception of the top layer. Consequently, all modeling languages that apply metamodeling are restricted to the definition of the top model in the metamodeling hierarchy. The top layer itself is often regarded to be its own meta-model. For graph based modeling, the mathematical formalization behind this relationship is called a graph homomorphism. To explain a graph homomorphism we need the notion of a directed graph.

Definition 1. A directed graph $G = (V, E, src, trg)$ is given by

- a collection $V$ of vertices,
• a collection $G_E$ of edges,
• a map $src^G : G_E \to G_V$ assigning each edge its source, and
• a map $trg^G : G_E \to G_V$ assigning to each edge its target.

We usually write $f : v \to u$ or $u \xrightarrow{f} v$ to indicate that $src^G(f) = v$ and that $trg^G(f) = u$.

To abbreviate, our graph $G$ has arrows (directed edges) and nodes (vertices). The source and target functions can take any arrow as input and will respectively return a source and a target node. Figure 2.2 shows an example of a simple graph with one vertex called $\text{Node}$ and one reflexive edge called $\text{Arrow}$.

![Figure 2.2: Directed graph example](image)

A graph homomorphism is a mapping between two graphs restricted by the following definition.

**Definition: Graph homomorphism.** A graph homomorphism $\varphi : \mathcal{G} \to \mathcal{H}$ between two graphs $\mathcal{G} = (G_V, G_E, src^G, trg^G)$ and $\mathcal{H} = (H_V, H_E, src^H, trg^H)$ is given by a pair of maps $\varphi_V : G_V \to H_V$ and $\varphi_E : G_E \to H_E$ such that for each edge $f : u \to v$ of $\mathcal{G}$ we have $\varphi_E(f) : \varphi_V(v) \to \varphi_V(u)$ in $\mathcal{H}$, i.e., we have $src^H(\varphi_E(f)) = \varphi_V(src^G(f))$ and $trg^H(\varphi_E(f)) = \varphi_V(trg^G(f))$ for all $f \in G_E$.

This definition of a graph homomorphism $\varphi : \mathcal{G} \to \mathcal{H}$ ensures that all nodes and arrows of a graph $\mathcal{G}$ are mapped to a corresponding node and arrow respectively while still maintaining the source and target functions. That is, given the nodes $u$ and $v$ in $\mathcal{G}$, $\varphi_V(u) = u'$ and $\varphi_V(v) = v'$ where $u', v' \in \mathcal{H}$, and for all functions $f : u \to v$ in $\mathcal{G}$, we are ensured that there exists a function $f' : u' \to v'$ in $\mathcal{H}$ given by $\varphi_E(f) = f'$. 

13
Let us use the trivial example from Figure 2.2, a graph $\mathcal{M}$ that contains one node $\text{Node}$ and one arrow $\text{Arrow}$. Let $\text{Arrow} : \text{Node} \rightarrow \text{Node}$ such that $\text{Arrow}$ is a reflexive arrow with $\text{Node}$ as both source and target. For any graph $\mathcal{G}$, for all nodes $n$ and arrows $a$, $\varphi_V(n) = \text{Node}$ and $\varphi_E(a = \text{Arrow})$ will ensure a valid graph homomorphism. This is proved by $\varphi_V(\text{src}^G(a)) = \text{src}^G(\varphi_E(\text{Arrow})) = \text{Node}$, $\text{trg}^G(\varphi_E(\text{Arrow})) = \varphi_V(\text{trg}^G(\text{Arrow})) = \text{Node}$. This relationship is visualized in Figure 2.3.

![Figure 2.3: Graph homomorphism example](image)

In the context of metamodeling, the term "typed by" is often used when talking about the graph homomorphism between a model and its metamodel. An element "A" is typed by "B" means that the element "A" in a model is mapped to the element "B" from the metamodel. This is how using graph homomorphisms defines the relationship between model and metamodel, $\varphi(A) = B$.

A DPF model is defined by specification. The specification consists of two parts, a graph and a signature. The graph contains basic modeling elements, named nodes and directed arrows.

**Definition: Specification.** A diagrammatic specification $\mathcal{G} = (S, C^\Sigma : \Sigma)$ consists of an underlying graph $S$ together with a set of atomic constraints $C^\Sigma$. The graph represents the structure of the model, and predicates from a predefined diagrammatic predicate signature $\Sigma$ are used to add constraints to this structure.
DPF uses a n-layered metamodel concept. Meaning that there can be an unlimited number of metamodels in a vertical typed by hierarchy. These layers are commonly named by the letter M and numbered by order. Generally, a layer $M_i$ is a model using the abstract syntax in layer $M_{i-1}$. A metamodels metamodel is often referred to as the meta-metamodel. The DPF modeling language in itself is defined in its abstract syntax with one node called $\text{Node}$ and one reflexive arrow called $\text{ Arrow}$ like in Figure 2.2. From this simple structure, users can define their own models over multiple layers to define their own modeling languages. Figure 2.4 shows an example of how a three layered metamodel hierarchy could look like in DPF. Layer $M_{n+2}$ shows the DPF defined metamodel. In $M_{n+1}$ a new model with three nodes and two arrows are defined. The dotted arrows shows the graph homomorphisms that makes up the metamodeling. In this case, we would say that $M_{n+1}$ is typed by the DPF metamodel in $M_{n+2}$. More specifically, we can say that the nodes A, B and C are typed by $\text{Node}$, while f and g are typed by $\text{ Arrow}$. The lowest layer consists of some model $M_n$ that only has two nodes, each with a typed-by relationship as well. If the graph of a model does not have a valid graph homomorphism to its type models graph, it would not be a valid DPF model.

Figure 2.4: DPF metamodeling example
This example shows how DPF uses metamodeling to define modeling languages. In itself, the DPF metamodel was used as abstract syntax when defining the model in $Mn + 1$. But $Mn + 1$ becomes the abstract syntax for $Mn$. This basically means that through metamodeling, we used the default DPF model as a language to define another modeling language. Section 2.4 illustrates how this method is used to define the DERF language.

### 2.3.2 Constraint Modelling

Constraint modelling goes beyond a pure graph based methods of defining modelling rules by applying semantic meaning to models. Looking back to the definition of the DPF specification, we had a set of atomic constraints $C^\Sigma$ that was based on a predicate signature $\Sigma$.

**Definition: Signature.** A diagrammatic predicate signature $\Sigma = (P^\Sigma, \alpha^\Sigma)$ consists of a collection of predicate symbols $P^\Sigma$ with a map $\alpha^\Sigma$ that assigns a graph to each predicate symbol $p \in P^\Sigma$. $\alpha^\Sigma(p)$ is called the arity of the predicate symbol $p$.

Figure 2.5 shows an example of a simple signature containing one predicate. All atomic constraints derives from predicates defined in a signature. An atomic constraint is a constraint added to a model element which is defined by a predicate. In this example, the predicate is $[\text{irr}]$ which is short for irreflexive. Predicates comes with both a visualization and a semantic interpretation. As shown in Visualization, the $[\text{irr}]$ predicate can be used in an atomic constraint attached to an arrow. This is called an arrow predicate. In DPF, there are many different kinds of predicates. For example, a predicate can be on a node or be between two arrows. Example of these kinds of predicates are described in section 2.4.2. The arity, $\alpha^\Sigma(p)$, of the atomic constraint is a shape that defines how this atomic constraint can be mapped to the graph. The arity needs to have a graph homomorphism to the graph of the specification it is attached to. The semantic interpretation, shown as set theory statement, defines the meaning of the $[\text{irr}]$ constraint. Figure 2.6 shows how the $[\text{irr}]$ predicate can be applied to a DPF model.
<table>
<thead>
<tr>
<th>p</th>
<th>$\alpha^\Sigma(p)$</th>
<th>Visualization</th>
<th>Semantic Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[irr]</td>
<td>f</td>
<td>f [irr]</td>
<td>$\forall x \in X : x \notin f(x)$</td>
</tr>
</tbody>
</table>

Figure 2.5: A sample signature for DPF \[32\]

Figure 2.6: DPF example of application of the [irr] constraint.

Figure 2.6 shows two DPF models where the bottom one at M(n) is typed by the upper one at m(n+1). The [irr] constraint on Arrow derives from the [irr] predicate. As shown on Figure 2.6, the arity, $\alpha^\Sigma([irr])$, has a valid graph homomorphism to the part of the model it is applied to. The semantic meaning of the constraint is applied to the graph of the model $M_n$. Both arrows $f$ and $g$ that are typed by Arrow fulfills the constraint as none of them are reflexive. If a constraint is broken, the model it applies to is no longer a valid DPF model.

To check a constraint, a pullback construction \[31\] is used. A pullback identifies the components by their graph homomorphisms. Figure 2.7 shows a visual example of this. There are four graphs in the figure. The arity of [irr], the graph $M_n$, the type graph $M_{n+1}$ and a result graph. The [irr] predicate components has a graph homomorphism to $M_{n+1}$ Node and Arrow. Further, A, B and f from $M_n$ also has a
graph homomorphism to $M_n$ Node and Arrow. The pullback result of this would be $A$, $B$ and $f$. This way we can identify the graph elements of $M_n$ which we need to check against the semantics of the $[\text{irr}]$ predicate.

Figure 2.7: A specification $M_n$, its type $M_{n+1}$ together with a pullback of the $[\text{irr}]$ predicate.

2.3.3 DPF Summary

Metamodelling based on typing and constraint satisfaction is the key concepts in DPF. DPF allows for an unlimited number of metamodel layers. Each model consists of an underlying graph decorated with a set of atomic constraints. Figure 2.8 shows how metamodelling works in DPF. Each layer $M$ represents one specification. The graph $S_n$ of a specification $\mathcal{G}_n$ is constricted by constraints based on predicates from the signature $\Sigma_{n+1}$ from the above specification $\mathcal{G}_{n+1}$ where the atomic constraint are added to the graph $S_{n+1}$.
2.4 The DERF Workflow Language

DERF [33, 34] is a formalized diagrammatic workflow modeling language. In DERF, activities are known as tasks and the directed arrows connecting them are known as flows. The name DERF comes from that tasks can be in one of four states: Disabled, Enabled, Running or Finished. Figure 2.9 shows an example of the previous activity diagram from Figure 2.1 as a DERF model. Note that in DERF, each flow has a name. But there are not any control structures in place yet. The rest of this chapter will look into how DERF is defined. Since DERF has been formalized upon DPF, we will first look at the metamodel-hierarchy and then move on to constraint modeling.

2.4.1 DERF - Metamodelling

DERF uses a three layer metamodel hierarchy below of the default DPF metamodel as shown in Figure 2.10. Level $M_2$ consists of one node Task and an arrow Flow. This is the DERF metamodel and it defines all valid models in the layer below.
Figure 2.9: A small hospital procedure represented in simplified DERF, based on [34]
This is the basis for the DERF language. The dotted arrows between the layers in Figure 2.10 shows the typed by graph homomorphism relationship. The graph homomorphism is also indication behind the name as an extra visual aid. In this case, all models based on the metamodel will be valid as long as they only consists of nodes of type Task and flows between these nodes of type Flow. As mentioned, DERF and DPF are closely related. DERF was originally created in DPF. DPF is therefore a natural choice to use when implementing DERF. The $M_2 -$ DERFMetamodel serves as the abstract syntax of DERF and can be directly typed by the DPF metamodel with $\iota_V(\text{Task}) = \text{Node}$ and $\iota_E(\text{Flow}) = \text{Arrow}$.

The next layer, $M_1 -$ DERFImplementation is where users can define their own workflow models. The last layer, $M_0 -$ DERFInterpretation will be referred to as the interpretation layer. Just like Java bytecode would be interpreted to machine code, a DERF implementation is interpreted down to the interpretation layer. But the example in Figure 2.9 does not include any control structures like the UML activity diagram example in Figure 2.1. This will be introduced in the following section 2.4.2.
2.4.2 DERF - Constraint Modelling

DERF applies constraints in two different manners. The first manner is language constraints. A language constraint is added to the abstract syntax to express restrictions on how an implementing model can be made. DERF uses one such constraints. It is visualized in Figure 2.11.
Figure 2.11: DERF $M_2$ Predicates

Figure 2.11 consists of three different parts. The **Predicate** is the concrete syntax that can be appended to a model element. In this case \([\text{irr}]\) is a short term for irreflexive. **Visualization** shows how this is done in an example. **Semantics (instances)** shows the semantic meaning in a diagrammatic style. In contrast to how the \([\text{irr}]\) predicate was presented in section 2.3.1, the arity is not shown. **Visualization** and the arity are exactly the same with the exception of the predicate labels. The **Semantics (instances)** demonstrates valid instances of the visualization. Here, an instance is a model that uses the **Visualization** as a metamodel. In this case, all flows between different tasks are legal. But the \([\text{irr}]\) constraint does not allow flow arrows to have the same source and target. For example, adding an arrow with $x$: **Task** as both source and target would render the **Visualization** model invalid. The DERF language uses this constraint in $M_2 - \text{DERF Metamodel}$ as shown in **Visualization**, disallowing reflexive arrows in $M_1 - \text{DERF Implementation}$.

The second set of constraints that DERF uses are control structure constraints. These constraints can be applied to the $M_1 - \text{DERF Implementation}$ layer. In the previous UML activity diagram (Figure 2.1) we had control structures such as the exclusive-or diamonds. DERF implements all these constraints and one additional. Namely \([\text{and}\_\text{split}], [\text{and}\_\text{join}], [\text{xor}\_\text{split}], [\text{xor}\_\text{join}]\) and \([\text{NodeMult}]\). As the \([\text{irr}]\) predicate was applied to the $M_2 - \text{DERF Metamodel}$ layer and created restrictions for the models at the $M_1 - \text{DERF Implementation}$ layer, these predicates are applied to the $M_1 - \text{DERF Implementation}$ layer and create restrictions on $M_0 - \text{DERF Interpretation}$. As already mentioned, the interpretation layer should be viewed as a runtime environment, which is why we consider these constraints to be control structures. Figure 2.12 illustrates the \([\text{and}\_\text{split}],[\text{and}\_\text{join}]\) constraints.
The $[\text{and}_{\ast}]$ predicates are intended to have a similar semantic as a solid bar in UML activity diagrams. From the Figure 2.12, we can see that an interpretation of the $[\text{and}_{\ast}]$ predicate includes both following flows and tasks. While the $[\text{and}_{\text{join}}]$ predicate ensures that both $x:X$ and $y:Y$ exists before $z:Z$ is added.
Figure 2.13: DERF $M_1$ Predicates $[\text{xor} \_\text{split}]$ and $[\text{xor} \_\text{join}]$

Figure 2.13 shows the two $[\text{xor} \_\text{*}]$ predicates. It illustrates which valid DERF Interpretation modes that can be created from the visualization example. But this example highlights that we are using arrow constraints. The $[\text{xor} \_\text{join}]$ predicate does not actually require that only either $x:X$ or $y:Y$ is present. The constraint only constrains the arrows $:f$ and $:g$, such that the two cannot be in the same runtime model. This means that a runtime model for the $[\text{xor} \_\text{join}]$ visualization model can, in addition to what is illustrated, could be both $\{x:X, y:Y, :f\}$ or $\{x:X, y:Y, :g\}$. The latter one is illustrated in Figure 2.14.

Figure 2.14: Valid example extension to the semantic interpretation of $[\text{xor} \_\text{join}]$

The last predicate for the $M_1$ – DERF Implementation layer is $[\text{NodeMult}]$. As the name implies, this is a node constraint. This predicate is added for allowing
loops in the DERF workflow models [34]. While the semantic interpretation in
Figure 2.15 is open, this predicate is only meant to constrain loops. Figure 4.1
which will soon be formally introduced, shows an example of a correct use of the
\([\text{NodeMult}]\) predicate. Note that this predicate also takes in a parameter \(n\) which
is used as a loop counter.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Visualization</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{NodeMult}, n])</td>
<td>([\text{NodeMult}, n])</td>
<td>(</td>
</tr>
</tbody>
</table>

Figure 2.15: DERF \(M_1\) Predicates \([\text{NodeMult}]\)

Figure 2.16 illustrate how \([\text{NodeMult}]\) may look like. As can be see in Figure
2.16 the graph homomorphism sign \(\iota\) is included. Always when looking at
relationships between different layers of metamodels, it is the graph homomorphism
rules that applies. As shown, we can get \(n\) instances of \(X\) on the \(M_0 - \text{DERF Interpretation}\)
layer when using the \([\text{NodeMult}]\) predicate.

Figure 2.16: Valid example extension to the semantic interpretation of \([\text{NodeMult}]\)

This concludes the constraints for \(M_0 - \text{DERF Interpretation}\). Figure 4.1
shows the earlier DERF example workflow model (Figure 2.9) when the introduced
constraints has been applied. Now the model has all the logic from the UML
activity diagram example (Figure 2.1), with the exception of a starting and an end
point. In Chapter 4 we will see that tasks without incoming flows are natural start
points. And tasks without outgoing flows are candidates for end points. Note how
the \([\text{NodeMult}]\) constraint is used to create a loop from Evaluation2 to TakeDrug
and back again through **Bloodtest2**. The `[xor_split]` predicate ensures that the **End** activity is never reached while the loop is initiated.

At this point, the $M_0 \rightarrow DERFInterpretation$ layer has many constraints to fulfill. The semantic instances of the $M_1 \rightarrow DERFImplementation$ constraints becomes a part of the method to interpret DERF models. The final five constraints before this subsection is concluded are the $M_0 \rightarrow DERFInterpretation$ constraints. These includes the constraints related to the [E]nabled, [R]unning and [F]inished states of tasks. In addition, there is the two arrow predicates `[f]` and `[t]`. These labels are used to show which flow that has enabled a task. If a flow labeled `[t]`, for true, it means that it is the flow that has been used to enable the flows target. For this thesis, these constraints are merely labels. When running a DERF model (as mentioned in Section 2.4) the tasks which they represents, can either be enabled, running or finished. And the flows can be either true or false. Figure 2.18 illustrates these predicates. Even though we are not looking deeper into the meaning of these predicates, their existence is essential when moving forward through the next chapter as they give tasks different states.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Visualization</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>[E]</td>
<td>[E]</td>
<td>Enabled</td>
</tr>
<tr>
<td>[R]</td>
<td>[R]</td>
<td>Running</td>
</tr>
<tr>
<td>[F]</td>
<td>[F]</td>
<td>Finished</td>
</tr>
<tr>
<td>[f]</td>
<td>[f]</td>
<td>False</td>
</tr>
<tr>
<td>[t]</td>
<td>[t]</td>
<td>True</td>
</tr>
</tbody>
</table>

Figure 2.18: DERF $M_0$ Predicates
Figure 2.17: A small hospital procedure represented in DERF.
Chapter 3

Model Transformation

This chapter demonstrates how DERF models can be executed based on model transformations.

3.1 Model Transformations in General

Nikulaus Wirths provides an intuitive explanation of how MDE works by comparing it to traditional software engineering as an equation.

'Data Structures + Algorithms = Programs' \[1\],

and the pure model driven equivalent is

'Models + Transformations = Software' \[5\].

Transformations are not a new phenomenon in software engineering. Programming languages are mere abstractions from actual executable code and depends on compilers and interpreters to transform code written in a language such as Java or C++ to machine code. In MDE, a model transformation is a process that generates a target model from a source model. Transformations has many applications ranging from automatic implementation of a model in code to migrating from one modeling language to another. They can be exploited to gain a pure MDE approach to many typical software engineering tasks and problems, such as source control, automated testing and refactoring. Source control could be achieved by saving alterations in models as transformations instead of saving the actual models. Automated testing can be achieved through so-called Model-to-Text (M2T) transformations that can

\[1\]The correct citation is 'Algorithms + Data Structures = Programs'.
take a model as input and, for instance, output JUnit tests. One of the major common elements of model transformation is that they are defined through a set of rules, often specified in an explicit transformation language. These rules are based on the metamodel of the source and target models. A more specific example of a model transformation would be to migrate a UML class diagram to an EER database schema. Here, both the input and output models have their own abstract syntax and metamodel. Since these are formal modeling languages, we can define rules on how such a transformation should happen. This migration example is called an exogenous out-place model-to-model transformation. Exogenous means that the input and output models conforms to two different metamodels, model-to-model describes that both the input and output of this transformation are models. That a transformation is out-place means that it has to create the output model from scratch. Figure 3.1 shows the concept of exogenous model transformations. It is in the Transformation Specification that the transformation rules would be defined. Model transformation rules are specified in model transformation languages. For this specific transformation, we can imagine that the transformation rules would map classes to entities, class attributes to entity attributes and references to relations. The circle represents some transformation engine that can interpret the rules and perform the transformation. Endogenous model transformations are in principal just a simplification of exogenous transformations. In endogenous transformations, the source and target model has the same metamodel.

Figure 3.1: (Exogenous) Model transformation concept

30
3.2 Transformation Rules for DERF

A model transformation consists of several components, source and target models and transformation engines and languages. The actual model transformation is defined through transformation rules written in a transformation language for a model transformation engine. In Chapter 2 we looked at the semantics of the DERF modeling language. The goal of this thesis is to apply state space exploration techniques to DERF. The $M_0$ – DERFInterpretation layer gives us instances based on the $M_1$ – DERFImplementation layer. But we are still missing how we can move from one DERFInterpretation instance to another. We want to be able to create some meaningful state transition system corresponding to the execution of DERF. Model transformations can create an execution environment for the semantics provided by the DERF predicates. For the rest of this chapter, we will look at how the semantics can be specified using transformation rules that can reflect the execution of a DERF implementation model based on the DERF concepts introduced in [33] and transformation concepts used in the Henshin project [39]. At the end of this section, there will be an example of how the transformation rules can be applied to the example model from Figure 4.1.

To define transformation rules we need a formalized language. This thesis will focus on transforming the graph of the models. That makes it natural to take a graph based approach to model transformations. To understand how this works we need to introduce a couple of model transformation concepts.

A model transformation takes in a source model and modifies it to the target model. A transformation rule consists of left-hand side (LHS) and right-hand side (RHS) graphs. The LHS graph defines a pre-condition for the rule and is used to find matching graph structures in the source model. The RHS graph defines a post-condition for the rule and specifies graph structures that should be produced for each matching graph structure from the LHS matching. This could be to delete (produce none) or alter the elements or create new ones. Graph homomorphisms are used to trace elements between LHS and RHS. Section 3.1 mentioned that a model transformation could be used to translate a class diagram to an EER diagram. A rule for this translation could use a LHS graph that matched class elements. While the RHS creates EER entities for each matching class. The transformation
engine would extract all the matches from the LHS and produce a model based on the source model and the alterations specified on the RHS. Figure 3.2 shows how the transformation works by applying a double pushout \[31\]. Graphs for common elements between LHS / RHS (\(C\)) and between Source / Target (\(C'\)) are created. All the arrows represent graph homomorphisms between these graphs. The Target graph is created by:

- Deleting all the matching elements from LHS, excluding those in \(C\).
- Creating all the matching elements from RHS, excluding those from \(C\).

![Diagram](image)

**Figure 3.2: Principle behind the double pushout approach \[3\]**

Application conditions are additions to the LHS and RHS matching. Application conditions can be divided into two subgroups: *Negative Application Conditions* (NACs) and *Positive Application Conditions* (PACs). NACs allow us to specify elements that should not be present in a model for a LHS matching. PACs are used to specify elements that should always be present in an LHS matching.

Now we can take a closer look at how the DERF semantics can be defined as transformation rules. The rest of this thesis will work on generating the *DERF Interpretation* layer from a model created in the *DERF Implementation* layer. \[33\] introduces coupled transformation rules for DERF. These coupled transformation rules formalize the semantics behind the transitions that is introduced in this chapter.
To start, Figure 3.3 shows how a simple transformation rule of how a task in the interpreted layer can go from Enabled to Running. All the examples are based on the DERF workflow example in Figure 4.1. The visualization of the transformation rules is divided into three parts. The expected input for the rule. The rule definition of LHS and RHS with application conditions and the output of the transformed input. In this rule, the node labeled $E$ is deleted from the input through RHS application conditions and a new copy of the node is created with the label $R$. Imagining this as a running instance, maybe a simulation, the transformation rule has taken us one step further in the execution of this model. The next step would be going from Running to Finished. This is not visualized, because it is the same method would be applied, only the labels would differ.
Now that a task can be transformed from enabled to running to finished, it is time to look how the other predicates can be translated into rules. But there is one part of the DERF semantic that has not yet been discussed. That is the semantics of a flow arrow without a predicate. Simply enough, it goes from one state to the next. To explain this transformation, we will define that whenever a task is \([F]\)inished a flow arrow with that task as a source can initiate its target task to \([E]nabled\). Figure 3.4 shows how this is done with a transformation. From here on, node types will not be explicit included in the node name.

![Figure 3.4: Rule: Simple Arrow](image)

Keep in mind that these rules uses both the DERF\textit{Implementation} layer and the DERF\textit{Interpretation} layer. Model transformations are usually defined at the meta-level. So to transform a DERF\textit{Interpretation} model we need to specify which kind of task that should be created. Like in this example, a DERF\textit{Interpretation} task is created that is typed by the DERF\textit{Implementation} task Bloodtest2. The actual name of the created node could be anything. If this was a rule for deleting an unnamed tasks typed by Bloodtest2, most transformation engines would delete
all nodes that fitted the description.

The \([\text{and\_split}]\) and \([\text{and\_join}]\) rules are similar in behavior as well. We need one rule for each of these predicates. Figure 3.5 visualize the \([\text{and\_split}]\) semantics as a transformation rule.

![Figure 3.5: Rule: [and\_split]](image)

The \([\text{and\_join}]\) would require two \([F]inished\) nodes with two arrows \(f\) and \(g\) labeled with the \([\text{and\_join}]\) predicate and having the same target. Then outputting the updated model with the target node as \([E]nabled\).

The same concepts are used for translating the rest of the DERF semantics into transformation rules. Figure 3.6 shows a simplified version of all the rules. Here, the input is on the left hand side and the output is on the right hand side. The
names are taken from Figure 4.1, but the rules are independent of which DERF implementation they are used on. Also the typed-by relationship for the arrows is not included to make the image more readable. \([\text{xor} \_\text{split}]\) and \([\text{xor} \_\text{join}]\) has two rules. This is because the semantics behind the predicates opens up for two different execution paths.

Figure 3.6: A simplified version of all rules.

Figure 3.7 gives an example of how the transformation rules can be applied to create a sense of progression in the \(M_0\) – DERFInterpretation layer based on the model from Figure 4.1. The natural starting point is InitialEvaluation since it is the only task without any incoming flows. It is set to \(E\)nabled and the transformation rules are applied and the progress evolves.
Figure 3.7: An example of transformation rules in use

The next chapter covers state space exploration. Transformation rules provide a way to explore the state space based on the semantic interpretation of DERF. By using the transformation rules, we also limit what kind of models that can occur as we saw in Figure 3.7. The Evaluation1 task was not enabled before both Bloodtest1 and MRI had finished. But this counterexample is a valid model in the DERF Interpretation layer based on the constraints alone. In section 2.4.2 we mentioned that the [xor_split] constraint does not exclude any tasks, only flows.
But there is nothing stopping us from defining the rules in such a way that the task that is connected to the excluded flow is never added. Another way to think about this is that the transformations adds, or defines, meaningful behavior based on the semantics.
Chapter 4

State Space Exploration and Model Verification

Chapter 4 highlighted the need for software testing. In MDE, a technique called state space exploration may be used when testing and verifying behavioral properties of models. This chapter introduces what a state space is, how it can be generated and how models can be verified based on state spaces.

4.1 State Space Exploration

A state space consists of a set of states and transitions between those states. Together, states and transitions forms a directed graph where the nodes are states and the arrows are transitions. The transitions represents events that can take us from one state to the other. As an example, we can imagine a game of chess. The initial state is when the board has been set up with all the chess pieces in their starting positions. From this state, many other states can be reached. Following the rules, the player controlling the white pieces always starts. From the initial state, there are 12 possible moves. One for each pawn and two for each knight. Each of these valid moves results in new states for the chessboard. The transitions from the initial state to each of these is the actual movement of a chess piece. Playing through the game will end up in many new states and possible transitions. But as the game is played and choices are made, only a part of the states will be reached. This is called a partial state space. If we would look at all the possible
moves throughout a game of chess, we would end up with a full state space. Note
that each move does not necessarily create a new state. If an equal state already
exists when performing a move, a transition is added that goes to the pre-existing
state. The actual process of creating the states is called state space exploration.
The definition below gives a more accurate definition of a state space.

**Definition: State space.** A state space [20] is a directed graph consisting of a
finite set of nodes \( S \) and a finite set of directed arrows \( T \). A state \( s \) represents a
single unique instance and a transition \( t \) represents a state change that goes from
a state \( s \) to another state \( s' \). Any state \( s' \) that is a target of a transition from a
state \( s \), is called a direct successor for \( s \). A transition \( t \) between two states is often
denoted as \( s \xrightarrow{t} s' \). If a state \( s' \) is reachable from a state \( s \), we can write \( s \xrightarrow{*} s' \),
which means that there exists at least one path going from \( s \) to \( s' \) containing zero
or more transitions \( t \). The function \( \text{reach}(s) = \{ s' \in S | s \xrightarrow{*} s' \} \) denotes the set of
all states reachable from \( s \). An initial state, the first state that the rest of the state
space originates from, is often denoted \( s_{\text{initial}} \) or \( s_0 \).

For DERF, we can generate a state space for a model in the \( M_1 - \text{DERFImplementation} \)
layer where each state is a model in the \( M_0 - \text{DERFInterpretation} \) layer and
represents progress in the implementation model. The transitions between these
states are the model transformations presented in the previous chapter. The
following section takes a deeper look at what is required to create an automatic
state space exploration tool for DERF.

### 4.2 Automatic State Space Exploration for DERF

To generate a state space for a model, we need to know the semantics of the
modeling language that is used. Chapter 2 introduced the DERF language along
with the semantics of the language. Chapter 3 described how the semantics of
DERF can be applied to perform an execution of a DERFImplementation model
in the DERFInterpretation level.

The model in Figure 4.1 shows a workflow for a small hospital process. Here
a patient first receives a medical evaluation followed by both a bloodtest and an
MRI. Then a second medical evaluation is conducted before the patient is brought
through either procedure A or B. After the procedure the patient gets a third
evaluation and might end up on taking a drug treatment. If the patient is on the
drug treatment he/she will have to have a new bloodtest and evaluation. This
process might be repeated up to five times, as specified by the $[NodeMult, 5]$ predicate.

To generate the state space, we need an initial state. In the chess game example
the initial state was when the board is first set up. For DERF, we can look at the
transformation rules specified in chapter 3. All the transformations that enables
states require that they have an incoming flow from a finished task. This means that
tasks without incoming flows cannot be enabled. A natural choice is therefore to
enable all tasks that have none incoming flows. In Figure 4.1 InitialEvaluation
is the only task that meets this requirement. To create the initial state we simply
need to create a model in the DERFInterpretation layer that is typed by Figure
4.1 and add InitialEvaluation marked as being enabled ($[E]$). Figure 4.3(a)
shows the model corresponding to the initial state.

From here, model transformations can be used to generate the rest of the state
space (i.e. other valid states specified in the semantic meaning of the model). Using
the transformation rule in Figure 3.3 will transform the model in Figure 4.2(a) into
the model that can be seen in Figure 4.2(b). By adding a directed arrow going
from $s_0$ to $s_1$ we now have a partial state space containing two states with one
transition as shown in Figure 4.3(b).

Continuing this process, by applying model transformations to state models, we
can generate all states for the model in Figure 4.1. Figure 4.2(c) shows the state
$s_2$. InitialEvaluation is transformed into an finished task. The next step is to
apply the transformation specified in Figure 3.5 resulting in $s_3$ (Figure 4.2(d)).

From $s_{10}$, shown in Figure 4.2(e) there are two possible transformations that
are semantically valid. Both of the exclusive-or transformations are applicable.
When situations like this occur, a split is created in the state space as shown in
Figure 4.3(c). The two states are visualized in Figures 4.2(f) and 4.2(g).
Figure 4.1: A small hospital procedure represented in DERF. [31]
Figure 4.2: Statespace states for Figure 4.1
Listing 4.1: Pseudo code for a BFS statespace generation algorithm

Listing 4.1 is an algorithm to compute a state space. It uses a breadth-first approach. The algorithm uses three collections. A Queue to store unprocessed states, a set of Nodes to store the states and a set of Edges that stores transitions between two states as a triplet (label, source → target). To initiate the algorithm the initial state is put in the queue. And as long as the queue is not empty, the next state to process is taken out of the queue. The successor(states) method is where transformation rules are applied to a state to generate all its successor
states. All the successors are added to the queue of unprocessed states if they are not already contained in the state space. Finally the edge between the state being processed and all its potential successors is created.

The full state space for the model in Figure 4.1 is shown in Figure 4.4. In $s_2$ the tasks Bloodtest1 and MRI are added. The state space splits here because there is no single rule defining which order the two tasks should change state from enabled to running and then to finished. But when $s_{10}$ is reached, the [and_join] predicate ensures that both tasks are finished before the Evaluation1 task can be added, and the two paths are joined.

After $s_{13}$ the [xor_split] and [xor_join] predicates creates two paths that will never join. But both these paths look similar because of the symmetry of the exclusive-or paths in the implementation model.

On the path $s_{20} \rightarrow s_{22} \rightarrow s_{24}$. Evaluation2 goes through the enabled, ready and finished process. This is repeated in $s_{44} \rightarrow s_{46} \rightarrow s_{48}$ because of the [NodeMult, 2] predicate.

In states $s_{54}$ and $s_{55}$ the End task has been finished and there are no more tasks to process.

The full state space consists of 55 states and 59 transitions while the model consists of 10 tasks and 12 flows.
4.3 Verifying Model Behavior

State space exploration is a great tool when verifying model behavior. Instead of checking specific corner cases, we can check behavioral properties that holds for any possible model state. Model behavioral properties can be split into two different groups: model dependent and model independent properties. Model dependent behavior concerns behavioral properties that are specific to a model implementation. For example, for the model from Figure 4.1, we might want to check that a patient never goes through both ProcedureA and ProcedureB. Model independent behavioral properties focuses on more abstract properties such as deadlocking.

The most common methods of verifying model behavior is through the use of
temporal logics such as linear-time temporal logic (LTL) and computation tree logic (CTL) and is referred to as model checking. Temporal logics formalize notions related to time, such as next, some time in the future and always. A predicate is a boolean function that in this context is usually used to check some properties of a single state. Pre- and postconditions are special kinds of predicates that checks properties before, pre, and after, post, an operation. While invariants are used to specify conditions that always must be true. For example, we could specify that for all cases where A happens, B must always follow but never C. But these kinds of logics are outside the scope of this thesis. Instead, we will focus on properties that can be directly extracted from the structure of state space graphs without using any complex logic formulas. More specific, the focus will lie on the reachability and home properties.

Reachability properties [19]

A reachability properties are concerned with checking if a state $s_i$ can be reached from another state $s_j$. That is, a state $s_j$ is reachable from a state $s_i$ if and only if there exists a path from $s_i$ to $s_j$ in the state space. Reachability properties can help in understanding and verifying the flow of a model. For example, in the model from Figure 4.1 we could define a reachability property that checks that a state including Bloodtest2 is always a possible outcome if the task TakeDrug was enabled. However, reachability properties does not ensure that a state $s_j$ will always be reached from $s_i$. There might be one or more paths from $s_i$ that never ends up in $s_j$. Depth-first search can be used to find a path in a graph. This algorithm has a running time of $O(|V| + |E|)$ where $V$ is the set of all states (vertices) and $E$ is the set of all transitions (edges).

From our example in Section 4.2, we can investigate for the path from the initial state to the last state where the entire process has ended. Reachability($s_0$, $s_{55}$) will return a path, thereby ensuring that a patient can be assigned a medical team. If we try the same for $s_{24}$, even though this path ends up in "End" being processed, there exists no path to the actual state $s_{55}$. Reachability($s_{24}$, $s_{55}$) would not return any paths.

Home Properties [19]

Home properties are concerned with checking if a state $s_{home}$ can be reached
from any reachable state. In other words, a state $s_{\text{home}}$ can always be reached and there is no state that does not have a path to that state. On the other hand, a home marking is never guaranteed to be reached. Like in our example from Section 4.2 there is not always a single home state. Therefore we define a home space as a set of states $s^{a}_{\text{home}} = S$ where the same properties counts, only we do not know which of the specific home states where a given path can end up. What is known is that there always exists a path from that state $s_i$ to a state in the home space. In the example from Figure 4.4 we would get $s^{a}_{\text{home}} = \{s_{34}, s_{36}, s_{54}, s_{55}\}$ is a home space.

This property is usually checked by looking for strongly connected components (SCC) in the state space. An SCC graph is acyclic and it is always possible to reach a SCC that has no outgoing edges. SCCs without any outgoing edges are called terminal SCCs and all the states in all the terminal SCCs makes up the home space.

4.4 State Space Explosion

State space explosion [19] is a common problem in state space exploration. State space explosion is the MDE equivalent of combinatorial explosion. The number of possible states grows very rapidly as the models complexity grows. The example used in this thesis has already demonstrated how a small and simple DERF model with only 10 tasks and 12 flows has a state space containing 55 states and 59 transitions. That is roughly five times the size in just nodes and arrows compared to the number of tasks and flows. The book 'Colored Petri Nets' [19] has some examples based on real life MDE applications. A model of an network transmit protocol, somewhat like the Transmission Control Protocol (TCP), is used to show that a state space can easily grow from 81 states and 100 transitions up to hundred of thousands of states with millions of transitions. This is without actually altering the model, but adding complexity through number of packets sent and upper limit for resending lost ones [19 p. 186]. Although the modeling language Coloured Petri Nets [19] (CPN) is more complex than the current state of DERF, it gives a good indication on how a state space can explode. What we can take from the CPN book is that the blast from the state space explosion grows larger with the complexity of the model.
Several methods have been developed to handle memory limitations that result from state space explosion. [23] introduces some of them. The symmetry reduction method, also explained in [19], that exploits observations that if two states are bisimilar then it is sufficient to only explore the successors of one of them. State compression that compresses state data and/or shares common components is another technique for alleviating the state space explosion problem. One method to do this could be storing states as vectors representing alterations from the initial state. Simpler, but not that efficient, is using secondary memory such as a magnetic disk drive to store data when the primary memory fills up. In this thesis we will explore an alternative called the sweep-line method.

4.5 The Sweep-line Method

Many typical behavioral properties can be checked-on-the-fly when exploring the state space. This opens up for deleting processed parts of the state space from memory before the exploration is completed. But to do this, we need some method to be sure that the states that are removed are not needed later in the process. Otherwise we might end up with a non-terminating state space exploration or only exploring a partial state space. This is where the sweep-line method [6, 19, 20] comes in. The sweep-line method is based on the concept of progress in the state space. One can imagine drawing lines through a state space, where all the states on the left side are processed and not necessary for the rest of the generation process, and on the right we find the states that need to be explored to find undiscovered states. Figure 4.5 shows an example of possible sweep-lines. The lines divide the state space into layers. When all the direct successors to the states of a layer has been generated, the states in that layer no longer has any purpose for the continuation of the exploration process, and can thus be deleted. Although states are deleted during the exploration, the sweep-line guarantees that all states are visited.
The sweep-line method utilizes progress measures to determine when to delete states and which states to delete. A progress measure specifies an ordering of the states in the state space. A progress value is assigned to each state such that it is greater than or equal to all of its predecessors. In the example in Figure 4.5, the progress values for the states is the number in their section. So when \(s_2\) is explored, the lower value states \(s_0\) and \(s_1\) can be removed from memory. [20] gives the following definition of a progress measure.

**Definition: Progress measure [20].** A progress measure is a tuple \(P = (O, \sqsubseteq, \psi)\) such that \(O\) is a set of progress values, \(\sqsubseteq\) is a partial order on \(O\), and \(\psi : S \rightarrow O\) is a progress mapping where \(S\) is a finite set of states. \(P\) is monotonic if \(\forall s, s' \in \text{reach}(s_1) : s \rightarrow^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')\). Otherwise, \(P\) is non-monotonic.

Even though the progress measures from the example in Figure 4.5 would help in memory reduction they are far from perfect. Figure 4.6 shows an optimal layer division for a sample space. For example, \(s_0\) is no longer required after \(s_1\) has been created. A good progress measure minimizes memory consumption as much as possible. There exists different strategies to find progress measures. They can be derived from analyzing a sampling at the state space itself and manually give states progress measures, the model in question, or from analyzing the semantics of the model. The latter option is the most versatile as we do not need to rethink the progress measures each time an alteration is introduced in the model. Further, this allows us to create an automatic testing environment using the sweep-line method. In DERF, this means looking at the \(M_2 - DERFMeta\) layer and the meaning of the model constraints that DERF uses. A source of progress is being provided by tasks being Enabled, going to Running and Finished. The order of which this
happens is given in the $M_I - DERFImplementation$ model through our model transformations for interpreting DERF. Suggestions for DERF progress measures is given in section 4.5.2.

Figure 4.6: An optimal sweep-line layer division.

The following listing is the sweep-line algorithm from [20].

```
1   Roots. Insert (s_initial)
2   Nodes. Insert (s_initial)
3   while ¬(Roots.Empty()) {start sweep} do
4       Unprocessed ← Roots
5       Roots ← ∅
6       Layer ← ∅
7       $\psi_c = \text{Unprocessed.GetMin()}$ {progress value for current layer}
8       while ¬(Unprocessed.Empty()) do
9           s ← Unprocessed.GetMinElement()
10          if $\psi_c \sqsubseteq \psi(s)$ {sweep-line moves} then
11             for all $s' \in \text{Layer}$ such that ¬ Nodes.Persistent($s'$) do
12                Nodes.Delete($s'$) {delete non-persistent states in current layer}
13             end for
14             Layer ← ∅
15             $\psi_c = \psi(s)$ {update progress value for current layer}
16         end if
```

51
The algorithm has some similarities to the other state space algorithm in Listing 4.1 but looks complex at first glance. The major difference is of course that it utilizes the progress measures to delete states from memory.

The sweep-line algorithm uses 4 collections to store states. States in $\text{Roots}$ are the topmost states and are the basis for each run of the sweep-line algorithm. $\text{Layer}$ keeps track of all processed states, states where the successors has been discovered, for the current sweep-line layer. The sweep-line layer consists of all the states that have the same progress measure. $\text{Nodes}$ keeps track of all states within a sweep-line, including the unprocessed ones, in addition to states that are marked as persistent. $\text{Unprocessed}$ holds all discovered but unprocessed nodes, regardless of which layer they belong to.

When the sweep-line algorithm starts, the initial states are put into $\text{Roots}$ and $\text{Nodes}$. Then the outer loop, starting at line 3, starts a sweep-line exploration on each iteration. Inside this loop all states that are in $\text{Roots}$ extracted to $\text{Unprocessed}$. $\psi_c$ represents the current layers progress value and the algorithm start with the lowest value of the unprocessed states.

At line 8 a second loop starts. This loop iterates as long as there exists unprocessed states. Here each unprocessed state $s$, starting with the lowest possible progress measure, is explored.
If the progress measure for the current state \( s \) that is going to be explored is larger than the current layer value, a new layer is started and the sweep-line is moved. This is done by deleting all non-persistent states in \( \text{Nodes} \) (lines 11 - 13), emptying \( \text{Layer} \) (line 14) and setting the layer value \( \psi_c \) to the progress measure of \( s \) (line 15).

Disregarding if the sweep-line has moved, \( s \) is then put into \( \text{Layer} \) as it is about to be explored. The new loop, line 18, finds and iterates over all the successors, \( s' \) of \( s \) that are not already contained in \( \text{Nodes} \). But if that would be the case, \( s' \) has already been explored and should not get re-explored.

If \( s' \) was not previously explored, is first inserted in \( \text{Nodes} \) at line 20 and then the sweep-line algorithm checks if \( s \xrightarrow{} s' \) is a regress edge at line 21. A regress edge is an transition where the source has a higher progress measure than the target. The normal state space exploration method has no problem with transitions going to states that are already explored. It is just a matter of checking if that state already exists in the state space. But since the sweep-line method can delete states with lower progress measures, there is no guarantee that the state is explored. The progress measure might be inconsistent, or non-monotonic. When this happens, \( s' \) is marked persistent in \( \text{Nodes} \) and put into \( \text{Roots} \) (line 21-24). But if \( s' \) does not have a lower progress measure than the layer it is inserted into \( \text{Unprocessed} \) at line 25 so it can be explored when it is one of the states with the lowest progress measure.

When the sweep-line algorithm runs out of unexplored nodes in \( \text{Unprocessed} \) we end back up to a new iteration of the outer loop. Here, the eventual regress edge states can start a new sweep-line exploration. The only information we want to keep is the persisted states that created regress edges. This way, we ensure that all possible states are found, but the exploration will not loop indefinitely as the same regress edges will not be explored twice. This is ensured by that persisted states are kept in \( \text{Nodes} \) and never removed. Therefor, when the sweep-line method finds a persisted state in a new run, it will not explore it again (line 11).

### 4.5.1 Verifying properties with the sweep-line method

On-the-fly state space exploration has some challenges when it comes to verifying properties as states are deleted before the entire state space is explored. Section 4.3
introduced in detail the reachability property and home properties. As explained in [6], these properties are easy to check with the sweep-line method.

Checking the reachability property is straightforward since the sweep-line method guarantees that all states are visited. To check if a state $s_j$ is reachable from a state $s_i$, $\text{reachable}(s_i, s_j)$, we simply need to add $s_i$ as the initial state. If $s_j$ is reachable from $s_i$ the sweep-line method will find a path. If $s_j$ is never found $\text{reachable}(s_i, s_j)$ is false.

The home property was defined as a terminal SCC. The sweep-line method ensures that all members of a SCC will be deleted at the same time. Therefore it is only a question to check if the SCC is terminal. If it is, all its states are members of the home space. But this is only correct for monotonic progress measures.

[6] and [13] has more on verifying properties with the sweep-line method.

4.5.2 Progress Measures for DERF

To create progress measures that can be used on any DERF Implementation model, we need to look to the DERF semantics and define progress measures based on them. Figure 4.7 shows the four DERF predicates $\text{and}_\text{join}$, $\text{and}_\text{split}$, $\text{xor}_\text{join}$ and $\text{xor}_\text{split}$. Based on these predicates we can propose a few different strategies for defining progress measures. Each strategy maps values to states on the DERF Implementation level and can use them to calculate progress values for their DERF Interpretation states.
Strategy 0:
Strategy 0 is a simplistic progress measure that maps all states to 0. All states get the same progress value of 0. Table 4.1 shows how Strategy 0 maps progress measures to the example from Figure 4.1.
<table>
<thead>
<tr>
<th>Node / Task</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitialEvaluation</td>
<td>0</td>
<td>1 (\rightarrow 1)</td>
</tr>
<tr>
<td>Bloodtest1</td>
<td>0</td>
<td>10 (\rightarrow 2)</td>
</tr>
<tr>
<td>MRI</td>
<td>0</td>
<td>10 (\rightarrow 2)</td>
</tr>
<tr>
<td>Evaluation1</td>
<td>0</td>
<td>100 (\rightarrow 4)</td>
</tr>
<tr>
<td>ProcedureA</td>
<td>0</td>
<td>1000 (\rightarrow 8)</td>
</tr>
<tr>
<td>ProcedureB</td>
<td>0</td>
<td>100000000000 (\rightarrow 2048)</td>
</tr>
<tr>
<td>Evaluation2</td>
<td>0</td>
<td>100000000000 (\rightarrow 4096)</td>
</tr>
<tr>
<td>TakeDrug</td>
<td>0</td>
<td>1000000000 (\rightarrow 512)</td>
</tr>
<tr>
<td>Bloodtest2</td>
<td>0</td>
<td>10000000 (\rightarrow 64)</td>
</tr>
<tr>
<td>End</td>
<td>0</td>
<td>100000000000 (\rightarrow 1024)</td>
</tr>
</tbody>
</table>

Table 4.1: Progress measure mappings for model in Figure 4.1

**Strategy A:**

Listing 4.3 shows the algorithm behind Strategy A. In Strategy A all tasks are given an increasing numeric value for their progress measure value. The value is based on the flows that connects the tasks. If the flow does not have any constraints, the target of the flow is then given the value of the flows source + 1. For an shape like \(X \rightarrow Y\), \(y\) is assigned the value of \(x + 1\). Given the [and_split] shape in Figure 4.7(a) both \(Y\) and \(Z\) is given the value of \(X + 1\). For arrows with the [xor_split] constraint, as in Figure 4.7(c), \(Y\) is given the value of \(X\). But instead of giving the same value to \(Y\), the strategy continues to go deeper into the model. When there are no more flows to follow, \(Z\) is given the value of the last task that has been mapped so far + 1. A mixture of breadth-first and depth-first graph traversing is used to implement the mapping algorithm.

Table 4.1 shows how Strategy A maps progress measures to the example from Figure 4.1. To calculate the progress value for a state, each enabled, running and finished task is looked up in the progress mapping and the values are added together. A state where InitialEvaluation is finished and Bloodtest1 and MRI would get the value \(0 + 1 + 1 = 2\).

```
1  ProgressMap<
2  ProgressMap.Put(Node_initial, 0)
3  Processed ← ∅
4  initializeProgressMeasure(Node_initial, Processed)
5  function InitializeProgressMeasure(Current, Processed)
```
if Current.OutgoingArrows == 0
    return Current

Processed ← ∅

for all A ∈ Current.OutgoingArrows
    if Processed.Contains(A)
        return Current

if A.Constraint == [and\_split]
    B ← GetOther(A)
    ProgressMap.Put(A.Target, progressMap.Get(Current) + 1)
    ProgressMap.Put(B.Target, progressMap.Get(Current) + 1)
    Processed.Add(A, Processed)
    Processed.Add(B, Processed)
    Current ← InitializeProgressMeasure(A.Target)
    Current ← InitializeProgressMeasure(B.Target)

else if A.Constraint == [xor\_split]
    B ← GetOther(A)
    ProgressMap.Put(A.Target, progressMap.Get(Current) + 1)
    Processed.Add(A, Processed)
    Current ← InitializeProgressMeasure(A.Target)
    ProgressMap.Put(B.Target, progressMap.Get(Current) + 1)
    Processed.Add(B, Processed)
    Current ← InitializeProgressMeasure(B.Target)

else
    if ¬ Processed.Contains(Current)
        ProgressMap.Put(A.Target, progressMap.Get(Current) + 1)
        Current ← InitializeProgressMeasure(A.Target, Processed)

return Current

Listing 4.3: Strategy A
Strategy B:

Strategy B is similar to Strategy A, but there are a few changes. While Strategy A mapped nodes to integer values, Strategy B maps nodes to binary strings. The method used to increment is similar, but now an increment means to add an extra zero at the end of the binary string. The method getBinaryForPredecessor(Node Current) fetches the binary representation for Current and returns the same string only with the extra zero attached. Listing 4.4 shows the full algorithm for mapping progress values. Again, Table 4.1 shows how Strategy B maps progress measures to the example from Figure 4.1. The progress value for a state is calculated by looking through all the binary strings. For each time we find a ‘1’ at an index $i$, there is added a ‘1’ to the same index in a result string. At the end, the result string is translated into base 10. A state where InitialEvaluation is finished and Bloodtest1 and MRI we would combine the strings ”1”, ”10” and ”10” and get the value ”11” which is equal to 3 in base 10.

```python
1 ProgressMap<Node, Integer>
2 ProgressMap.Put(Node_initial, 1)
3 ProcessedArrows ← ∅
4 initializeProgressMeasure(Node_initial, ProcessedArrows)
5
6 function InitializeProgressMeasure(Current, ProcessedArrows)
7    if Current.OutgoingArrows == 0
8        return Current
9
10    ProcessedArrows ← ∅
11
12    for all A ∈ Current.OutgoingArrows
13        if ProcessedArrows.Contains(A)
14            return Current
15
16    Processed . Add(A)
17
18    if A.Constraint == [and, split]
19        B ← GetOther(A)
20
21        ProgressMap.Put(A.Target, getBinaryForPredecessor(Current))
22        ProgressMap.Put(B.Target, getBinaryForPredecessor(Current))
23
24    ProcessedArrows.Add(A, ProcessedArrows)
```
\begin{lstlisting}[language=Java]
  ProcessedArrows.Add(B, ProcessedArrows)
  Current ← InitializeProgressMeasure(A.Target)
  Current ← InitializeProgressMeasure(B.Target)
  
  else if A.Constraint == [xor_split]
  B ← GetOther(A)

  ProgressMap.Put(A.Target, getBinaryForPredecessor(Current))
  ProcessedArrows.Add(A, ProcessedArrows)
  Current ← InitializeProgressMeasure(A.Target)

  ProgressMap.Put(B.Target, getBinaryForPredecessor(Current))
  ProcessedArrows.Add(B, ProcessedArrows)
  Current ← InitializeProgressMeasure(B.Target)

  else
    if ¬ ProcessedArrows.Contains(Current)
      ProgressMap.Put(A.Target, getBinaryForPredecessor(Current))
      Current ← InitializeProgressMeasure(A.Target, ProcessedArrows)

  return Current
\end{lstlisting}

Listing 4.4: Strategy B
Chapter 5

Implementation

This chapter introduces the tools, technologies and techniques used to implement the prototype used for experimenting with the foundations laid down by this thesis. DERF is implemented in the DPF Workbench, Henshin and state space exploration is combined and the sweep-line method is implemented.

5.1 Overview

Figure 5.1: Overview of experiment implementation

Figure 5.1 shows an overview of the prototype implementation. EMF and ECore provides the foundations to the entire stack. Both the DPF Workbench and the Henshin Transformation Tool are implemented on EMF and ECore. In this prototype, the DERF language and DERF State space Explorer are implemented.
The DERF language is implemented with the DPF Workbenches graphical user interface.

The DERF State space Explorer is implemented in Java. It utilizes the DPF Workbench to define the DERF models and the Henshin Transformation Tool to transform models. EMF's ECore implementation is used to create transformation rules with Henshin.

5.2 Implementing the DERF Language

This section introduces how to implement DERF in the DPF Workbench. But first we need to look at EMF and ECore as it is used to implement the DPF Workbench.

5.2.1 EMF and ECore

Eclipse Modeling Framework (EMF) is a modeling framework originally based on Meta Object Facility (MOF) and is implemented in Eclipse EMF provides tools for building Eclipse Plugin tools through metamodeling and code generation. MOF is a standardization for MDE developed by the Object Management Group (OMG). The current standard of MOF that EMF uses is called Essential MOF (EMOF). It is a graph based, four layered metamodeling environment consisting of a meta-metamodel, metamodel, models and realization of the models. The meta-metamodel is itself called EMOF. Similar to the default DPF metamodel, EMOF is a general purpose modeling language and provides the foundation for all other models based on it as well as it is typed by itself. OMG uses EMOF to define many different modeling languages on the metamodel layer of the EMOF environment, including their standardization of UML. According to Ed Merks, a project leader of EMOF, "ECore is the defacto reference implementation of OMG’s EMOF". EMF has many similarities to how DERF is implemented over DPF. In EMF, the meta-metamodel layer is the ECore model. Users can create their own modeling languages on the metamodel layer. These models are then used to create model implementations based on the user language.

Figure 5.2 shows a simple model implemented in ECore using EMF. Person, Student, Teacher and Course are EClass elements. These elements has a range
of different attributes which are different kinds of \textit{EAttribute}, including \texttt{Persons\_first\_name} attribute which is an \texttt{EString}. The edges are of types \textit{EReference} and \textit{Inheritance}.

![Figure 5.2: The visual EMF ECore editor showing a simple model created in ECore.](image)

\section*{5.2.2 DPF Workbench}

DPF was introduced in \ref{5.2}. The DPF Workbench \cite{22} was originally presented in a master thesis by Øyvind Bech \cite{4}, and provides a modeling environment for DPF build on \textit{Eclipse Modeling Framework} (EMF) \cite{9} and \texttt{Ecore} \cite{10} technologies. The DPF Workbench has several features, including tools for creating model specifications and constraint signatures. The \textit{DPF Model Editor} is a graphical editor that is used to create and edit specifications. The \textit{DPF Signature Editor} is used to create and implement diagrammatic constraints that can be used with any specification. The semantics of the constraints can either be implemented in Java or the \textit{Object Constraint Language} (OCL) \cite{27}.

To create a new specification, both a metamodel (as an existing specification) and
a signature must be added. The DPF Workbench provides both a default metamodel and signature. The default metamodel consists of the simple shape shown in Figure 5.3 as described in section 5.2.2. Since all models are required to have a metamodel, when going up the metamodeling hierarchy of a DPF specification, it will always end up on the default metamodel. In other words, all models implemented in the DPF Workbench are at some level typed, therefore restricted, by the default metamodel. The default signature includes arrow constraints such as injective, surjective and irreflexive. Figure 5.4 shows how the workbench is used to create a multi-metamodeling hierarchy. Notice the similarities with the DPF meta-hierarchy shown in Figure 2.8.

![Figure 5.3: The DPF Metamodel.](image)

![Figure 5.4: Metamodeling with the DPF Workbench.](image)

The DPF Specification is implemented as ECore models. All elements in a DPF Signature and DPF Specification are typed over ECore elements such as EClass, EReference and EAttribute. For example, the DPF Arrow and DPF Node are typed...
by EClass with the attributes id and name typed by EAttribute. EReferences are used to connect the different elements. For example, connect DPF arrows with DPF nodes, one for source and another for target. Figure 5.5 shows an extraction from the underlying ECore implementation of DPF. Each node is an EClass, the arrows are EReferences and the attributes are EAttributes. This extraction demonstrates both how a DPF specification contains a graph with nodes and arrows as well as how the typing system is implemented. EReferences whose names end with s indicates that there can be any number of edges. In other words, a graph can contain any number of nodes and arrows.

Figure 5.5: An extract of the DPFCore model showing graphs, nodes, arrows and typing.

Figure 5.6 shows how constraints are implemented for nodes. Like in Figure 5.5, EReferences names implies multiplicity. The only exception here is the edge nodemapping between GraphHomomorphism and NodeToNodeMap. A GraphHomomorphism can have multiple NodeToNodeMaps. Most of the EAttributes are self-explanatory. The predicate symbol is the label that is used to identify the predicate, ie [and_split].

64
Figure 5.6: An extract of the DPFCore model showing constraints and predicates on Nodes.

Figure 5.7 shows how constraints are implemented for arrows. The same principle applies, only NodeToNodeMap is exchanged with an ArrowToArrowMap. Also, in the DPF Workbench, arrow constraints are attached to both the arrow(s) and the nodes that are the source(s) and target(s) of the arrow(s).

Figure 5.7: An extract of the DPFCore model showing constraints and predicates on Arrows.
5.2.3 Implementing DERF in the DPF Workbench

DERF was originally created as a DPF implementation and is therefore easy to implement in the DPF Workbench. Figure 5.8 shows how DERF fits in a DPF hierarchy, excluding graph homomorphisms and constraint application arrows. The default DPF metamodel is in the top layer with graph homomorphisms $\varphi_V(\text{Task} = \text{Node})$ and $\varphi_E(\text{Flow} = \text{Arrow})$.

![Figure 5.8: DERF implementation in DPF](image)

Figure 5.9 illustrates an example of how the DPF Signature Editor is used to implement constraints. The list to the left shows all the constraints that belongs to
the signature. The right hand side of the signature editor is where the constraint is specified. For this example the \texttt{[and\_split]} predicate is selected. The visualization field specifies how the predicate should be visualized. The \texttt{[and\_split]} predicate is an “ArrowToArrow” predicate, meaning it will be visualized as an label connected to two arrows. The arity of the constraint is specified in ”Graph Details”. The details for the field ”Validator” is hidden in the figure, but this is where the semantics of the constraint is specified.

![Signature Predicates](image)

Figure 5.9: Using the DPF Workbench Signature Editor to implement DERF constraints.

Listing 5.1 shows an extraction of the semantic implementation of \texttt{[and\_split]}, demonstrating how the arrows can be extracted in. When the arrows are extracted it is just a question to check that the semantics of the constraint is not violated. To do this, we can check that for each arrow in the \texttt{ab} list, the source node has a corresponding arrow in the \texttt{ac} list. Note that the names used in the listing corresponds to those in the arity.

```java
1  MapProgressMeasure(DERFModel) {
2      for all t ∈ Tasks
3          if t.GetOutgoings == 0
4              subroutine(t);
5      end if
```
end for

}

Sub(t) {
    for all f : t.GetOutgoings do
        if (f.hasPredicate(PREDICATE_A))
            NewValue = f.target.Value + 1
        f.target = f.GetTarget
        c = Sub(f.target)
        NewValue = c.Value + 1
    }
Listing 5.1: Extraction of and_split constraint implementation.

5.3 Implementing the Model Transformations

Model transformations are created and executed with the Henshin Transformation Tool [39]. To different approaches has been tested in this prototype.

5.3.1 The Henshin Project

One of the goals of this thesis is to use transformations rules to interpret the DERF semantics, and create models representing states in the DERFInterpretation layer based on DERFImplementation models. We have looked at the semantics of DERF in Chapter 2 and how to translate this to transformation rules in Chapter 3. The Henshin transformation tool has been chosen for this task since it, like the DPF Workbench, is implemented for the EMF and based on ECore.

The Henshin Project is an official model transformation tool for EMF. The Hensin Editor was developed as a student project at the Technical University of Berlin in 2010. It was continued in a bachelor thesis by Johann Schmidt [36] and a master thesis by Angeline Warning [42]. Henshin uses a graph-based transformation language to define transformation rules. Each transformation rule consists of a
5.3.2 Defining transformation rules

Henshin comes with a graphical transformation rule editor, providing users with an intuitive method to define transformation rules. Transformation rules are created in Henshin Diagram files that can be found using the Eclipse wizard. When a Henshin Diagram file has been created, elements from the model language(s) of the source and target models can be imported through the context menu ”Import Package...”. Henshin uses a graph-based transformation language where rules consists of LHS and RHS graphs describing models by their underlying structure. The default elements consists of nodes, edges and attributes, corresponding to the Ecore EClass, EReference and EAttribute. Other elements based on ECore, like the DPFCore model, can be imported and used when defining transformation rules. Figure 5.10 shows what a transformation rule in the graphical editor looks like. Here, the DPFCore model has been imported and is used to create the graph representing the rule.

Other elements based on ECore can be imported into Henshin, allowing for them to be used when defining transformation rules. Figure 5.10 shows what a transformation rule in the graphical editor looks like. DPFs core elements has been imported and are used to create the graph representing the rule.

Other elements based on ECore, like the implementation of DPF in the DPF Workbench, can be imported. Figure 5.10 shows what a transformation rule in the graphical editor looks like. DPFs core elements has been imported and are used to create the graph representing the rule.
The rule in Figure 5.10 demonstrates how the simple arrow transformation rule from Figure 3.3 can be translated into a Henshin transformation rule. In this rule there are a number of nodes, edges and attributes, all corresponding to an element in the DPF Core model. The DPF Core model is where all the DPF elements are implemented. The rule specifies that we have a specification. The specification has a graph containing the nodes TakeDrug and Bloodtest2 and the arrow \( f_9 \). Each of the three elements has a typed-by relation, typeArrow or typeNode, to the corresponding tasks from the DERFImplementation which is the metamodel for DERFInterpretation. The arrows has source (TakeDrug) and target (Bloodtest2) edges. In addition, there are constraints on each node and arrow that is connected to a predicate. In this rule, we require that TakeDrug has a constraint using the \([F]\) predicate and we add the constraints \([t]\) and \([E]\) to \( f_9 \) and Bloodtest2 respectively. The attributes, node names, arrow names and predicate symbols, helps identifying the elements in the graph. All the elements in this transformation rule corresponds to the underlying ECore structure in the DPF Core model. Note that the example in Figure 5.10 has been simplified to make it readable. Any elements from the source model that are not specified are automatically marked as \( \ll \text{preserve} \gg \) by Henshin as long as they, or any arrows
connecting them, are not affected by the transformation. Bear in mind that this example excludes several required elements. For example, the DPFCore model extraction in Figure 5.6 shows that constraints uses \textit{GraphHomomorphism} and \textit{NodeToNodeMap} to connect the predicates \textit{Shape} nodes to the \textit{Graph} nodes. Also, the fact that the arrow predicate is attached to the nodes as well, as discussed in section 5.2.2. These dependencies must be handled in the actual implementation.

In addition to all the elements, the rule contains two different application conditions: \(
\ll \text{preserve} \gg\) and \(
\ll \text{create} \gg\). Both of these are positive application conditions. The preserve conditions tells Henshin that these elements should be on both the LHS and RHS graph. This ensures that the elements are both present in the input and are added to the output model. The positive application condition \(
\ll \text{create} \gg\) places elements on the RHS graph so they will be created.

The negative application conditions that are interesting for this thesis are the \(
\ll \text{forbid} \gg\) and \(
\ll \text{delete} \gg\) conditions. \(
\ll \text{forbid} \gg\) can be used to define elements that should not be in the model if a rule is applied. For example, we can create a rule that forbids any incoming arrows to a node. The \(
\ll \text{delete} \gg\) condition is used to remove elements from the model.

The forbid condition comes in handy when implementing the \([\text{xor}^*]\) constraints. Looking back at Figure 2.13, the \([\text{xor\_split}]\) predicate can be transformed into two different transformation rules. Both rules would use \(x:X\) as input. One of the rules would add \(\overset{1}{f}\ y:Y\) and forbid \(\overset{2}{g}\), the other would add \(\overset{2}{g}\ z:Z\) and forbid \(\overset{1}{f}\). This way, the \([\text{xor\_split}]\) constraint is fulfilled since the transformations would not be applicable if the opposite arrow already exists in the model.

The delete condition can be used to implement the transformations that takes a DERF task from \([E]\text{nable}\) to \([R]\text{unning}\) and from \([R]\text{unning}\) to \([F]\text{inished}\). Figure 5.12 illustrates a partial implementation of this rule in Henshin.
Henshin will generate the LHS and RHS graphs based on these application conditions. Preserve elements are added to both the LHS and RHS as they must be present before the transformation and after. Delete elements are added to the LHS as they must be present, but are not on the RHS and will not be added to the output model. Create elements are only added to the RHS. The forbid condition is a special condition. It is added on the LHS as a nested condition. In this case, the nested condition is an additional match pattern to the LHS graph and it is required that the match fails.

The actual creation of these rules are done programmatically in the implementation. The DPFCore model (as it is implemented in ECore) consists of many nodes and edges and the graphical editor is cumbersome to use when the transformation rules consists of many elements. For example, the simple_arrow rule actually contains 36 nodes and 71 edges. Henshin provides an API for creating rules through the HenshinFactory class. With this class, we can create the exact same graphs as we would create in the graphical editor. But there are some limiting factors when using the Henshin API. The largest disadvantage with using Henshin programmatically is the poor, and sometimes lack of, documentation. This often leads to unexpected behavior, since some transformation specifications must be implemented in a specific order. If the order is broken, adding new elements to the rule graph may override behavior specified earlier.
5.3.3 Transforming Models

To run a rule as a model transformation in Henshin, the rule must first be contained inside a transformation unit. A transformation unit defines the order of how the rules are applied. If a transformation consists of more than one rule, the output from a preceding rule will be used as input for the next. Henshin has a set of different transformation units to choose from, such as IndependentUnit, PriorityUnit and ConditionalUnit to mention some. IndependentUnits can be used when there is no dependency between the rules and it does not matter which order they run in. PriorityUnits allows users to define the order in which the rules should be applied to the target model. ConditionalUnits uses an if-else pattern to choose between two different rules. Units can also be nested. For example, given two PriorityUnits A and B, we can use a ConditionalUnit to choose which unit should be applied. If the transformation only consists of one single rule, the rule itself can be used as a unit. The unit can then be put into a Henshin module which is supplied to the Henshin transformation engine with the source model.

5.3.4 Using Henshin to transform DERF models

As mentioned in the introduction to this section, two different approaches have been used to interpret the DERF semantics. The first one, Approach A, follows the semantics as they were introduced in [33] and [34]. The second one, Approach B, extends these definitions of the DERF language but relies on far fewer transformation rules. These two methods will also be further explored in Section 5.4 where the consequences of each approach is further investigated.

5.3.5 The [NodeMult, n] predicate

A problem that has not been discussed yet is how to implement the transformation rule associated with [NodeMult, n] predicate. Since DERF is still a very basic language, we have no method to distinguish each loop. One solution is to generate multiple copies of the afflicted tasks. Another is to extend the model with a counter attribute that can be used to check against the [NodeMult, n] parameter. Both of these methods have some disadvantage. The first method, creating n copies of
each task and flow in the loop, can lead to a clustered graph as a result. While the second method requires that we extend the DERF semantics and add the counter attribute at the DERFInterpretation layer.

This prototype expands the DERF models on the DERFInterpretation layer by adding multiple copies of the loop nodes. For instance, based on the example in Figure 4.1, additional copies of Evaluation2, TakeDrug and Bloodtest2 are created.

5.3.5.1 Approach A

Approach A follows the definitions of DERF and its semantics as they have been introduced this far. When a Task is disabled it is not present in the model. To add the task, we need to create a new node in the model that represents that task, as shown in Figure 5.10. The Bloodtest2 task is created and typed to the corresponding type. Bloodtest2 lives in DERFInterpretation while the type lives in DERFImplementation. Due to a limitation in Henshin, this cannot be implemented in a more abstract method. For the DERFInterpretation model to make sense, we need to include each tasks name. The best solution would be to have a single rule defined over DERFMetamodel that said if some task $X$ in DERFInterpretation is [F]inished, and in DERFImplementation its type has some outgoing flow $f$ to some other task $Y$ without any arrow-to-arrow constraints, add the corresponding flow and task in DERFInterpretation. The problem with this approach is that we don’t know the names of the flow and task we want to add. And Henshin does not have any methods for looking up the names of the types and copying them.
5.3.5.2 Approach B

For Approach B, the prototype introduce an new addition to DERF, a disabled predicate \([D]\) as show in Figure 5.13. It is questionable to how new this is, as Disabled is a part of the DERF name. But it has not been introduced in [33] and [34].

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Visualization</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>([D])</td>
<td>X</td>
<td>Disabled</td>
</tr>
</tbody>
</table>

Figure 5.13: \([D]\) predicate in DERF \(M_0\)

The additional predicate from Figure 5.13 allows us to having disabled tasks in the \(DERFInterpretation\) layer. This opens up for a more general approach to defining the DERF semantics in Henshin. Figure 5.14 shows the updated version of the \(simple\_flow\) rule. The task (\(Bloodtest2\)) and flow (\(f9\)) we had be create in Approach A already exists in the model. All that is needed in this approach is to say we have some node that is connected \([F]\) predicate, an arrow that is connected to the \([f]\) predicate and yet another node connected to the \([D]\) predicate. Since we are working on \(DERFInterpretation\) models, by definition we know that all these nodes and arrows are typed by \(Task\) and \(Flow\) two layers above. The edges going from the arrows constraint to the \([f]\) predicate is replaced with an edge to the \([t]\) predicate. And the edge between the target nodes constraint is moved from the
[D] predicate to the [E] predicate.

Figure 5.14: Alternative partial implementation of the simple arrow transformation from Figure 3.4 in Henshin

Note that the transformation rule in Figure 5.14 is simplified and missing several components like the example in Figure 5.10.

5.4 Automatic State space Exploring Using Henshin

So far we have implemented the DERF language in the DPF Workbench and looked at how the DERF transformation rules can be translated into Henshin rules. This section will look into how we can combine these things with the state space exploration algorithm in Listing 4.1 from Chapter 4. There has been implemented two solutions for this thesis based on the approaches introduced in section 5.3.1. But both approaches uses the same base implementation.

To start with, the algorithm from Listing 4.1 must be implemented. A graph structure is used to represent the actual state space. The graph vertices are a set of states represented by a State class while the edges are implemented as a map where the key is the original state and the value is its successor. In the State class, the state model is stored as a DPF Specification. Listing 5.2 shows a minimal Java interface of the State implementation.

76
public interface IState {
    public int[] setVectorRepresentation(int[] vector);
    public boolean equals(State s);
    public Specification getStateSpecification();
}

Listing 5.2: Interface for statespace state representation.

Each time a new state is found it is added to the state space graph. But we need something to prevent storing duplicate states. To solve this, the state space graph uses a HashSet to store the states. A state is defined by its DERFInterpretation models nodes and arrows, with their respective constraints. And each state implements the hashCode() method by using HashCodeBuilder [14] from the Apache Commons library [2]. The hashcode is build on the contents of the DPF specification contained in the state. More specific, every task, node and constraint from the DERFInterpretation.

The edges representing transitions are implemented as pairs of references to State instances, such that the pair $<s_i, s_j>$ represents an transition $s_i \xrightarrow{t^*} s_j$. The queue is implemented as a simple first-in-last-out queue implementation.

The actual state space exploration algorithm is implemented in the class StateSpaceExplorer. It requires two parameters, the initial DERFImplementation model that should be explored and a list of Henshin rules. The StateSpaceExplorer uses Henshin transformations to implement the successors(state) method. Given some state $s_i$, successors($s_i$) it applies all available transformation rules to the state.

In addition to the transformation rules defined in Chapter 3 we also need some initial transformation rule(s) that takes enables the first node(s). As argued, a good approach is to enable tasks without any incoming flows. But by taking this approach, there must exist at least one task without any incoming flows in the DERFImplementation model.

The differences between the two approaches is the list of transformation rules that are used and the initial DERFInterpretation model that is given to the StateSpaceExplorer. Each approach requires its own solution for an automatic state space exploration.
5.4.1 Approach A

For Approach A, an algorithm has been implemented that goes through the DERFIImplementation model that is going to be explored and generates all the transformation rules needed to generate all the states, including the initial rule(s). But there are a couple of exceptions. The transformation rules taking a state from \([E]\) to \([R]\) and from \(R\) to \(F\) does not need to be generated for each state. The problem in Approach A is that we don’t know the names of the tasks and flows when they are added to the DERFIInterpretation. This is not the case for the two mentioned rules as they don’t require for any new tasks or flows to be created.

The initial DERFIInterpretation model is just an empty model. This follows that in the initial state no tasks should be enabled. And disabled states in Approach A are non-existing. The DPF Workbench supplies methods to create specifications and add nodes, arrows, constraints, etc.

5.4.2 Approach B

Since Approach B only uses rules that are independent from the DERFIImplementation models details, the transformation rules for this approach are already predefined. But this approach requires the initial DERFIInterpretation model to contain all the necessary tasks and flows labeled with the \([D]\) and \([f]\) predicates.

The following rules are predefined. Take note that all the following rules also handles the flow predicates \([f]\) and \([t]\)

- \(\text{InitialD2E}\) - Looks for tasks without any incoming arrows and enables them.
- \(\text{E2RRule}\) - Transforms enabled tasks to ready.
- \(\text{R2FRule}\) - Transforms running tasks to finished.
- \(\text{SimpleArrowRule}\) - As explained in section 5.3.5.1
- \(\text{AndJoinRule}\) - Enable the targets if the source is finished.
- \(\text{AndSplitRule}\) - Enables the target tasks if the source task is finished.
• *XorJoinRule* - Enables target only if exactly one of the source tasks are finished.

• *XorSplitRule* - Enables one of the targets only if the source is finished and the other target is disabled.

• *XorLoopExir* - A special case of the XorSplit rule used to exit loops.

### 5.5 Implementing the Sweep-line Algorithm

Implementing the sweep-line algorithm requires a few changes from the previous state space generation method in section 5.4. First we need to switch the state space algorithm with the sweep-line state space algorithm. Then we need to implement a method to calculate progress measures for states.

#### 5.5.1 The sweep-line algorithm

The sweep-line algorithm is specified in Listing 4.2. The implementation makes use of two datastructures. HashSets with states is used to represent Nodes and Layer. Roots and Unprocessed uses a priority-queue implementation that ensures the state with the lowest progress measure is first out. In addition, the State class has been modified to contain the progress measure value for the given state. The progress measure value is stored as a normal integer after it has been calculated.

#### 5.5.2 Progress measures

The implementation of progress measures uses a mapping function that maps tasks from the *DERFImplementation* layer to values. This mapping is then used to calculate a states progress measure by applying the mapping to the states *DERFInterpretation* model. The suggested progress measures are already introduced in section 4.5.2. For each new *DERFImplementation* model we want to explore, we need to do a new progress measure mapping. The sweep-line implementation uses this mapping to calculate the progress value for each state.
Chapter 6

Evaluation and Results

This chapter introduces the results from the prototype and evaluates them. It also introduces how to get the implementation and tools to test the prototype.

6.1 Using the prototype

The prototype can be downloaded through bitbucket at https://bitbucket.org/endre86/master-submission. It consists of Eclipse projects created in Eclipse Luna. Everything that is necessary to run the prototype is supplied.

To create models, the DERF metamodel and signature is supplied in the /resources/derf/ folder of the state space exploring project. In addition, you need to download the DPF Workbench. Download instructions can be found at http://dpf.hib.no/downloads/.

6.2 Test models

Five methods has been included in the test of the sweep-line method. Each of these models are present in the testmodels folder.
6.2.1 Model A

Model A is the same model that has been used as an example through this thesis and is shown in Figure 4.1. It combines all the features of the DERF modeling language in a small model.

6.2.2 Model B

Model B is attached Figure 8.1 and consists of random elements to create a larger state. This is done by introducing an early $xor\_split$.

6.2.3 Model C

Model C, attachment Figure 8.2 is one big loop. It starts with a MS Computer Science student that has to choose some practical classes. The practical classes consists of assignments and ends with an exam. If the student fails the exam, he has to start over and choose a class.

6.2.4 Model D

Model D, attachment Figure 8.3 is yet a random model, this time created to test how the progress measures can handle $and_*$ predicates.

6.3 The Normal State Space Exploration Method

6.3.1 Comparing Approach A and Approach B

Exploring DERF state spaces by using model transformations has been successful, resulting in full state spaces. But time consumption is a major limiting factor. Each atomic rule, for example the transformation $InitialEvaluation[E] \rightarrow InitialEvaluation[R]$, are applied to each state. Listing 6.1 is output from a state space exploration of a small DERF graph. Even though there are only 95 states in the state space and 18 transformation rules, the total amount of transformation attempts is 1710. From the output we can see that about 87% of the time used on transformations gives no result. This percentage grows with the size of the DERFImplementation model.
Number of transformation rules used: 18

State space generated:

States: 95
Edges: 191

Total transformations: 1710
Total time used on transformations: 6396 ms
Average time per transformation: 3 ms
Longest transformation time: 409 ms

Total successful transformations: 191
Total time used on successful transformations: 805 ms
Average time per successful transformation: 4 ms
Longest successful transformation time: 132 ms

Failed transformations: 1519
Time used on unsuccessful transformations: 5591 ms
Average time per unsuccessful transformation: 3 ms
Longest unsuccessful transformation time: 409 ms

Average total time per successful transformation: 33 ms

Total time to generate state space (ms): 6403.0

Listing 6.1: Output from statespace exploring

Listings 6.2, 6.4, 6.3, 6.5 shows the output for Approach B. In comparison, Approach B uses much less time on failed transformations in comparison to the state space size.
Average time per transformation: 4 ms
Longest transformation time: 175 ms

Total successful transformations: 77
Total time used on successful transformations: 1204 ms
Average time per successful transformation: 15 ms
Longest successful transformation time: 175 ms

Failed transformations: 445
Time used on unsuccessful transformations: 920 ms
Average time per unsuccessful transformation: 2 ms
Longest unsuccessful transformation time: 120 ms

Average total time per successful transformation: 2 ms

Total time to generate state space (ms): 2350.0

Listing 6.2: Output from Approach A on Model A

Number of transformation rules used: 18

State space generated:
States: 88
Edges: 150

TIME STATISTICS:
Total transformations: 792
Total time used on transformations: 3101 ms
Average time per transformation: 3 ms
Longest transformation time: 170 ms

Total successful transformations: 154
Total time used on successful transformations: 1175 ms
Average time per successful transformation: 7 ms
Longest successful transformation time: 170 ms

Failed transformations: 638
Time used on unsuccessful transformations: 1926 ms
Average time per unsuccessful transformation: 3 ms
Longest unsuccessful transformation time: 105 ms
23 Average total time per successful transformation: 1 ms
24
25 Total time to generate statespace (ms): 3123.0

Listing 6.3: Output from Approach A on Model C

1 Number of transformation rules used: 18
2 State space generated:
3
4 States: 445
5 Edges: 1065
6
7 TIME STATISTICS:
8 Total transformations: 4005
9 Total time used on transformations: 22407 ms
10 Average time per transformation: 5 ms
11 Longest transformation time: 428 ms
12
13 Total successful transformations: 1115
14 Total time used on successful transformations: 16604 ms
15 Average time per successful transformation: 14 ms
16 Longest successful transformation time: 428 ms
17
18 Failed transformations: 2890
19 Time used on unsuccessful transformations: 5803 ms
20 Average time per unsuccessful transformation: 2 ms
21 Longest unsuccessful transformation time: 126 ms
22
23 Average total time per successful transformation: 4 ms
24
25 Total time to generate statespace (ms): 22505.0

Listing 6.4: Output from Approach A on Model B

1 Number of transformation rules used: 18
2 State space generated:
3
4 States: 295
5 Edges: 754
TIME STATISTICS:

Total transformations: 2655
Total time used on transformations: 21803 ms
Average time per transformation: 8 ms
Longest transformation time: 398 ms

Total successful transformations: 895
Total time used on successful transformations: 12430 ms
Average time per successful transformation: 13 ms
Longest successful transformation time: 162 ms

Failed transformations: 1760
Time used on unsuccessful transformations: 9373 ms
Average time per unsuccessful transformation: 5 ms
Longest unsuccessful transformation time: 398 ms

Average total time per successful transformation: 4 ms

Total time to generate statespace (ms): 21868.0

Listing 6.5: Output from Approach A on Model D

6.4 The Sweep-Line Method

6.4.1 Progress measure strategy results

The following tables shows how effective the progress measure strategies introduced in section 5.5.2. Each strategy is tested against the same set of test models. The headings Nodes and Edges represents how many nodes and edges there is in the full state space. Sweep-line peak is the maximum of nodes in memory while running the sweep-line algorithm. This includes unprocessed but discovered nodes, nodes in Layer and the nodes stored as roots for future sweep-line exploration. Node Ratio represents the relationship between the sweep-line peak and the actual number of states in the state space.

Strategy 0
<table>
<thead>
<tr>
<th>Model</th>
<th>Nodes</th>
<th>Edges</th>
<th>Sweep-line peak</th>
<th>Node Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model A</td>
<td>58</td>
<td>191</td>
<td>58</td>
<td>1.00</td>
</tr>
<tr>
<td>Model B</td>
<td>445</td>
<td>61</td>
<td>445</td>
<td>1.00</td>
</tr>
<tr>
<td>Model C</td>
<td>88</td>
<td>150</td>
<td>88</td>
<td>1.00</td>
</tr>
<tr>
<td>Model D</td>
<td>295</td>
<td>1065</td>
<td>42</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 6.1: Results for Strategy 0

As expected from a monotone progress measure, Strategy 0 does not limit the memory usage. In fact, for Model 2, more states were stored in memory than there were total in the state space. This comes from the nature of the sweep-line algorithm and how it handles back-edges as described in section 4.5.

Strategy A

<table>
<thead>
<tr>
<th>Model</th>
<th>Nodes</th>
<th>Edges</th>
<th>Sweep-line peak</th>
<th>Node Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model A</td>
<td>58</td>
<td>46</td>
<td>4</td>
<td>0.79</td>
</tr>
<tr>
<td>Model B</td>
<td>343</td>
<td>343</td>
<td>9</td>
<td>0.96</td>
</tr>
<tr>
<td>Model C</td>
<td>95</td>
<td>82</td>
<td>82</td>
<td>0.93</td>
</tr>
<tr>
<td>Model D</td>
<td>42</td>
<td>69</td>
<td>234</td>
<td>0.79</td>
</tr>
</tbody>
</table>

Table 6.2: Results for Strategy A

Strategy A was the first attempt of implementing progress measures and was originally designed to fit the model from Figure 4.1.

Strategy B

<table>
<thead>
<tr>
<th>Model</th>
<th>Nodes</th>
<th>Edges</th>
<th>Sweep-line peak</th>
<th>Node Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model A</td>
<td>9</td>
<td>191</td>
<td>45</td>
<td>0.77</td>
</tr>
<tr>
<td>Model B</td>
<td>13</td>
<td>61</td>
<td>428</td>
<td>0.96</td>
</tr>
<tr>
<td>Model C</td>
<td>95</td>
<td>150</td>
<td>88</td>
<td>1</td>
</tr>
<tr>
<td>Model D</td>
<td>42</td>
<td>1065</td>
<td>0.81</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Table 6.3: Results for Strategy B

Strategy B seems to give the best peak of the bunch for Model A. Other than that it does not give any phenomenal results.
Strategy 0 is not surprisingly the biggest looser. But both Strategy A and Strategy B have their weaknesses against Model B. None of these progress measure strategies results in any great
Chapter 7
Conclusions and Future Work

This chapter sums up the work done in this thesis and presents alternatives for future improvements and extensions.

7.1 Conclusions

In this thesis we have explored how to create a state space explorer for workflow modeling languages by using model to model transformations. We have also seen on one method to limit memory consumption during the state space exploration. The technologies and methods that has been used are the diagrammatic workflow modeling language DERF, DPF and the DPF Workbench, the Henshin Project and the sweep-line state space exploration method. The first part of the thesis introduced how metamodeling and diagrammatic constraints were used to define the semantics of DERF. Metamodelling defined typed-by relationship that made it possible to implement DERF as a complex multilevel language in DPF. Diagrammatic constraints were used to restrict the DERF such that a flow could not have the same task as both source and target. They were also used as control structures and extending the meaning of a task by giving it the enabled, running and finished predicates. These semantics were then translated into model transformation rules so we could create an interpreter for DERF. The transformations added a behavioral feature to the DERF models. And Henshin proved to be a good fit with the DPF Workbench since they both are implemented in ECore as extensions to the EMF.
The next step of the thesis was to create an automatic state space generator. By using model transformations, we could generate successor states under the state space exploration. The Henshin Project provided us with an API allowing us to generate model specific transformation rules through some predefined methods. But a big downside to this method was the amount of time used on unsuccessful transformation attempts. The majority of the time used to explore the state space was used on failed transformations. Therefore another approach was created that allowed us to create more abstract rules. This approach, Approach B, cuts drastically down on time used to generate state spaces for DERF.

We also introduced the state space explosion problem. The term state space explosion is used about the often exponential growth of states in the state space based on the complexity of the model that is explored. The sweep-line method was visited to look at a solution for this problem and three strategies for implementing progress measures for DERF were introduces. The sweep-line method showed good promise to be able to limit the amount of states kept in memory. But the progress measures still have a long way to go.

7.2 Future Work

There is still more work to be done before we have a satisfying state space exploration tool for DERF.

7.2.1 Improving transformation rule approach

As mentioned above, the implementation uses a lot of time on unsuccessful model transformations. A more general approach could be taken to defining rules. By defining rules based on the DERFMetamodel, instead of creating a transformation for each possible step in the interpretation of the DERFImplementation, model the number of failed transformation attempts would decrease significantly.

7.2.2 Create better progress measures

Even though only three progress measures were presented in Chapter 5, other approaches has been tested. These approaches has tried to tackle the \([NodeMult, n]\)
predicate. But there still is some work left on them before they render useful results. A good progress measure should be able to delete most of the state space that comes before the \([NodeMult, n]\) loop is activated. After an loop has been entered, the state space will branch out based on the number of times the loop is repeated. A possible solution can easily be introduced in the progress mapping. If an \([NodeMult, n]\) task is found, the exit task can be found and be given a high value (compared to the other tasks). This way, when a loop has finished, we can remove the entire loop tree. This solution can be problematic in combination of regress edges.

7.2.3 Verifying Model Behavior

For this thesis, the actual model verifications has not been implemented as the focus has been on state space exploring. Section 4.3 introduces the methods to verify two behavioral properties, reachability and home properties. Section 4.5.1 briefly introduces how these two properties can be checked with the sweep-line method. But the home properties method requires a monotonic progress measure. \[6\] and \[13\] provides a much more detailed look at how the sweep-line method can be used to check behavioral properties.

7.2.4 Creating an integrated DERF tool

As the DERF language grows, a collection of features should be gathered to work with DERF models. Not only state space exploring, but also tools for code generation and model transformations. The DPF Workbench has already been integrated with these tools. An future DERF toolset can be created as an extension to the DPF Workbench.
Bibliography


93


Chapter 8

Attachments
Figure 8.1: Test model B
Figure 8.2: Test model C