Preface

My study of Computer Science and Engineering started in the year 2008 at the Eindhoven University of Technology. During the 2010 spring-semester, I followed the course Software Evolution in which I was introduced to the concepts of Model-Driven Engineering (MDE). Since I recognized the possibilities of this relatively new software engineering paradigm, I decided to start following the related course Generic Language Technology and the seminar of the Software Engineering and Technology (SET) group. During the seminar, it became clear to me that the ideal topic for my graduation project would be in the field of MDE. Therefore I contacted the head of the SET group, Prof. Dr. Mark van den Brand. Together with PhD candidates Ir. Luc Engelen and Ir. Marcel van Amstel we discussed several topics and eventually decided to perform a study on quality assessment of model transformations based on graph transformation. We agreed that this would be a good starting point for my graduation project, with enough directions to expand in. Later during the project the focus was shifted to the tooling part that is involved in quality assessment.

I owe many thanks to my supervisor Prof. Dr. Mark van den Brand for giving me the opportunity to work on a very interesting graduation project and for his advices and feedback throughout the project. Special thanks goes to my tutor Ir. Marcel van Amstel for his daily guidance and feedback on my work. I wish him the best of luck in acquiring his doctorate. Furthermore, I thank Dr. Alexander Serebrenik and Dr. Serguei Roubtsov for taking place in my examination board.
Abstract

The complexity of today’s software systems is continuously growing, while their required level of reliability remains high. This trend, which is expected to continue in the near future, is confronting industry and academia with severe challenges. A promising approach to address these challenges is Model-Driven Engineering (MDE). MDE is a software engineering paradigm in which models play a central role throughout the entire development process. Models abstract from implementation details, which increases their focus on strategic architectural issues. This abstraction promotes models with a focus on a particular problem domain. Model transformation provides a mechanism for creating target models, based on information contained in existing source models. Because of their prominent role in today’s and future software engineering, there is the need to define and assess the quality of model transformations. One of the key technologies to assess software artifacts is software metrics. This thesis presents the results of a study on quality assessment of model transformations based on graph transformation. We defined a set of metrics for assessing the internal quality of model transformations specified in the Viatra2 model transformation language. Furthermore, we related these metrics to quality attributes based on our expectations. Since manual extraction of metrics is error prone, and becomes infeasible as the size of a model transformation grows, tools are usually developed to perform this task in an automated fashion. Currently, these tools are tailored towards the formalism at hand. Since their development is a time consuming task, we present a generic approach for the automated extraction of metrics from an arbitrary model (transformation). In addition, we developed a tool that puts the approach into practice. Its suitability is validated using a case study.
# Table of Contents

1 Introduction 3  
  1.1 Motivational Background ........................................ 3  
  1.2 Project goal ....................................................... 4  
  1.3 Thesis Outline .................................................... 5  

2 Related Work 7  
  2.1 Software Measurement ........................................... 7  
  2.2 Quality Assessment of Model Transformations .................. 7  
  2.3 Automated Quality Assessment .................................. 8  
  2.4 Thesis Contribution ............................................... 9  

3 Viatra2 11  
  3.1 Basic Concepts ................................................... 11  
  3.2 Model Transformation based on Graph Transformation ........ 12  
  3.3 Viatra2 Framework ................................................ 13  
  3.4 Viatra2 Model Transformation Language ....................... 13  
    3.4.1 Viatra Textual Metamodeling Language .................... 13  
    3.4.2 Viatra Textual Command Language ......................... 15  

4 Metrics Definition 19  
  4.1 Viatra2 Model Transformation Assessment ...................... 19  
  4.2 Quality Attributes .............................................. 19  
  4.3 Viatra2 Model Transformation Metrics .......................... 20  
    4.3.1 VTCL File Metrics ........................................... 20  
    4.3.2 GT Pattern Metrics .......................................... 20  
    4.3.3 GT Rule Metrics ............................................. 21  
    4.3.4 ASM Rule Metrics ........................................... 24  
    4.3.5 ASM Function Metrics ...................................... 24  
    4.3.6 Dependency Metrics ........................................ 26  
  4.4 Relating Metrics to Quality Attributes ....................... 28  
    4.4.1 Understandability ........................................... 28
## Table of Contents

4.4.2 Modifiability .................................................. 29  
4.4.3 Reusability .................................................... 29  
4.4.4 Modularity ...................................................... 30  
4.4.5 Completeness ................................................... 30  
4.4.6 Consistency ..................................................... 30  
4.4.7 Conciseness .................................................... 31  
4.5 Observations ...................................................... 31  

5 Towards a Generic Approach 33  
5.1 Traditional Approach ................................................. 33  
5.2 Intermediate Generic Representation ................................ 34  
5.3 No Intermediate Representation ..................................... 35  
5.4 Semantic Enrichment ................................................ 35  
5.5 Observations ...................................................... 37  

6 Semantic Enrichment 39  
6.1 Semantic Metamodel ............................................... 39  
6.2 Construct .......................................................... 40  
6.2.1 Construct Semantics ............................................. 40  
6.2.2 Construct Metrics ............................................... 40  
6.2.3 Construct Application .......................................... 41  
6.3 Id ................................................................. 42  
6.3.1 Id Semantics ................................................... 42  
6.3.2 Id Metrics ........................................................ 42  
6.3.3 Id Application .................................................. 43  
6.4 Counter ............................................................ 43  
6.4.1 Counter Semantics .............................................. 43  
6.4.2 Counter Metrics ................................................ 44  
6.4.3 Counter Application ............................................ 45  
6.4.4 Counter Specialization: Cyclomatic Complexity .............. 45  
6.5 Dependency Reference .............................................. 46  
6.5.1 Dependency Reference Semantics ................................ 46  
6.5.2 Dependency Reference Metrics .................................. 46  
6.5.3 Dependency Reference Application ............................ 47  
6.6 Tree Reference ..................................................... 48  
6.6.1 Tree Reference Semantics ..................................... 48  
6.6.2 Tree Reference Metrics ........................................ 48  
6.6.3 Tree Reference Application .................................... 49  
6.6.4 Tree Reference Specialization: Dummy Tree Reference ....... 49
Table of Contents

7 Automated Metrics Extraction
7.1 Metrics Extractor .......................................................... 51
  7.1.1 Aggregation Techniques ........................................... 52
  7.1.2 Metrics Metamodel .................................................. 52
  7.1.3 Implementation Techniques ..................................... 53
7.2 Viatra2 Model Generator ................................................ 54
  7.2.1 Viatra2 Metamodel .................................................. 54
  7.2.2 Implementation Techniques ..................................... 55

8 Validation ........................................................................... 59
  8.1 Validation Approach ..................................................... 59
  8.2 Semantic Enrichment of Viatra2 Metamodel ...................... 60
    8.2.1 Construct Associations ......................................... 60
    8.2.2 Id Associations .................................................... 60
    8.2.3 Cyclomatic Complexity Associations ....................... 60
    8.2.4 Dependency Reference Associations ....................... 60
    8.2.5 (Dummy) Tree Reference Associations .................... 61
  8.3 Results ........................................................................... 61
    8.3.1 Combination of properties ..................................... 61
    8.3.2 Unused parameters ................................................. 62
    8.3.3 Recursive GT patterns ............................................ 62
  8.4 Observations ............................................................... 62

9 Conclusions and Future Work .......................................... 63
  9.1 Conclusions ............................................................... 63
  9.2 Future Work ............................................................... 65

Bibliography ........................................................................ 67

A Metrics Extractor Guide ................................................. 71

B Viatra2 Model Generator Guide ...................................... 73

C Viatra2 Model Transformation: Activity Diagrams to Petri Nets 75
List of Figures

3.1 Context of model transformation .......................... 11
3.2 Context of graph transformation .......................... 12
3.3 Example of VTML file ...................................... 14
3.4 Specification of UML class diagram and corresponding GT patterns .................................. 14
3.5 Example of VTCL file ...................................... 15
3.6 Example of GT pattern ...................................... 16
3.7 Example of GT rule ......................................... 17
3.8 Example of standalone ASM rule .......................... 17
3.9 Example of ASM function .................................. 18

5.1 Architecture of tool according to traditional approach ................................................. 33
5.2 Architecture of tool using intermediate generic representation .................................... 34
5.3 Architecture of tool without intermediate representation ............................................. 35
5.4 Architecture of tool using semantic enrichment ......................................................... 36
5.5 Associating model element to semantic classes .......................................................... 36

6.1 Semantic metamodel .......................................... 39
6.2 Context of construct .......................................... 40
6.3 Application of construct ..................................... 41
6.4 Context of id .................................................. 43
6.5 Application of id .............................................. 44
6.6 Context of counter ........................................... 44
6.7 Application of counter ....................................... 45
6.8 Context of dependency reference ......................................................... 47
6.9 Application of dependency reference ......................................................... 47
6.10 Context of tree reference .................................... 49
6.11 Application of tree reference ................................ 50

7.1 Architecture of metrics extractor ................................................. 51
7.2 Metrics metamodel .......................................... 53
7.3 Example of metrics model .................................. 53
List of Figures

7.4 Architecture of Viatra2 model generator ........................................ 54
7.5 Viatra2 metamodel (1) ......................................................................... 56
7.6 Viatra2 metamodel (2) ......................................................................... 57
A.1 User interface of metrics extraction tool ............................................... 71
C.1 Activity Diagrams to Petri Nets (1) ......................................................... 75
C.2 Activity Diagrams to Petri Nets (2) ......................................................... 76
C.3 Activity Diagrams to Petri Nets (3) ......................................................... 77
C.4 Activity Diagrams to Petri Nets (4) ......................................................... 78
C.5 Activity Diagrams to Petri Nets (5) ......................................................... 79
C.6 Activity Diagrams to Petri Nets (6) ......................................................... 80
List of Tables

4.1 VTCL file metrics related to quality attributes .................................... 20
4.2 GT pattern metrics related to quality attributes .................................... 22
4.3 GT rule metrics related to quality attributes ......................................... 23
4.4 Standalone ASM rule metrics related to quality attributes ....................... 25
4.5 ASM function metrics related to quality attributes ................................. 26
4.6 Dependency metrics related to quality attributes (1) ............................... 27
4.7 Dependency metrics related to quality attributes (2) ............................... 28
Chapter 1

Introduction

This chapter describes the motivational background of the project and states its goal. The last section gives the outline of the remainder of this thesis.

1.1 Motivational Background

The complexity of today’s software systems is continuously growing, while their required level of reliability remains high. This trend, which is expected to continue in the near future, is confronting industry and academia with severe challenges. On one hand, software systems are becoming increasingly important in modern society. On the other hand, traditional software engineering has shown to be unable to keep up with this rapid growth of software demand and complexity. Developers are required to pay such close attention to implementation related details, that they often cannot focus on strategic architectural issues, such as system-wide correctness and performance [33].

One of the best ways to deal with software complexity is through the use of abstraction, problem decomposition and separation of concerns [34]. To this end, Model-Driven Engineering (MDE) [33] has been proposed. MDE is a software engineering paradigm in which models play a central role throughout the entire development process. It combines Domain-Specific Languages (DSLs) for modeling software, with model transformations for synthesizing them. Models abstract from implementation details, which increases their focus on strategic architectural issues. This abstraction promotes models with a greater focus on a particular problem domain. In addition, models are well suited for analysis techniques, e.g., model checking, which positively affect the reliability of software systems. Models are frequently expressed in DSLs. In contrast to General-Purpose Languages, e.g., Java, C++ and C, a DSL is a small, usually declarative language that offers expressive power focused on a particular problem domain [46]. Hence, it enables software developers to model using domain concepts rather than concepts provided by existing formalisms, which typically do not provide the required abstractions.

Model transformation is considered the heart and soul of MDE [34]. It provides a mechanism for creating target models, based on information contained in existing source models. Since the execution of model transformations can be automated, they can significantly improve development productivity and quality. Similar to traditional software artifacts, model transformations have to be used by multiple developers, maintained according to changing
Chapter 1: Introduction

requirements and should preferably be reused. Because of their prominent role in today’s and future software engineering, there is the need to define and assess their quality [41].

The quality of model transformations can be viewed in two different ways [38]. The internal quality of a model transformation is the quality of the transformation artifact itself, whereas its external quality concerns quality aspects that are observable from the outside [39], i.e., the quality change it induces on a model. Furthermore, a distinction can be made between direct and indirect quality assessment. When one wants to define and assess the internal quality of a model transformation, direct quality assessment is considered the appropriate approach [38].

One of the key technologies to assess software artifacts is software metrics. A software metric is an objective measure of some property of a software artifact. Software metrics have been studied extensively in the context of traditional software engineering [31, 12, 22]. In recent studies, however, metrics have also been applied in the context of MDE. In [27], the authors defined metrics for assessing the quality of models specified in the Unified Modeling Language (UML) [14]. Furthermore, metrics have been proposed for the quality assessment of model transformations [40, 51, 29, 43]. The observations obtained from these latter studies could eventually be used to propose a methodology which, if adhered to, leads to high-quality model transformations [40].

1.2 Project goal

This section describes the goal of the project by stating a number of research questions. Furthermore, it describes the activities that should be performed to answer these questions. The first research question that we aim to answer is stated as follows:

RQ1. Can we apply direct quality assessment, as proposed in [38], to assess the internal quality of model transformations based on graph transformation?

Answering this question requires focusing on a representative model transformation formalism that is based on graph transformation. In [6], we presented the results of a comparative study between the graph-based model transformation formalisms AGG [11], GrGen.NET [15], Henshin [1] and Viatra2 [47]. By relating language constructs to quality attributes, we indicated the extent to which the constructs are interesting with respect to direct quality assessment as proposed in [38]. Since Viatra2 appeared to be the most extensive formalism, we concluded that it can also be considered the most suited formalism for answering RQ1.

Since manual extraction of metrics is error prone, and becomes infeasible as the size of a model transformation grows, tools are usually developed to perform this task in an automated fashion. Currently, automated metrics extraction is performed by tools tailored towards the formalism at hand [40, 42, 29]. Since the development of these tools is a time consuming task, and most of them have a considerable amount of overlap in functionality, it is interesting to find out whether this task can be performed in a more generic fashion. The second research question that we aim to answer is therefore stated as follows:

RQ2. Can we realize automated metrics extraction for model transformations using a generic approach?
A generic approach can only be considered useful if it can (largely) replace the traditional approach of metrics extraction, i.e., developing a tool tailored towards the formalism at hand. Therefore, we need to determine to what extent the generic approach is appropriate for the extraction of a previously defined set of metrics. The third, and last, research question is therefore stated as follows:

**RQ3.** To what extent can the generic approach, as stated in RQ2, replace the traditional approach of automated metrics extraction?

To answer the research questions stated above, the following activities should be carried out:

1. Study the specification [48] of the Viatra2 model transformation language and identify a set of metrics that is suited for the assessment of model transformations specified in this language (RQ1).
2. Relate the identified metrics to quality attributes. To this end, various approaches can be applied, such as the Goal-Question-Metric approach [4] or a more intuitive approach [40, 43, 29] (RQ1).
3. Study the possibilities to generalize automated metrics extraction for model transformations and propose a generic approach (RQ2).
4. Determine suitable techniques for the implementation of the generic approach (RQ3).
5. Implement a metrics extraction tool according to the chosen approach and techniques (RQ3).
6. Validate the suitability of the generic approach by performing a case study. To this end, the set of metrics could be used that we identified in activity 1 (RQ3).

### 1.3 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 discusses the current state of the art of the research related to this work. Chapter 3 briefly presents a number of preliminary concepts related to model transformation in general, and graph transformation in particular. Furthermore, it introduces the Viatra2 model transformation language. The following five chapters can be considered the core of this work. Chapter 4 presents the work that we performed related to research question RQ1. It describes the metrics that we defined for assessing the internal quality of Viatra2 model transformations. In addition, it describes how we related these metrics to quality attributes. In the following two chapters we aim to answer research question RQ2. Chapter 5 proposes a generic approach for the automated extraction of metrics. This approach is elaborated on in Chapter 6. The two subsequent chapters describe the work related to research question RQ3. Chapter 7 describes a metrics extraction tool that implements the generic approach. The validity of this approach is presented in Chapter 8. The last chapter, Chapter 9, presents the conclusions of the project and directions for further research.
Chapter 2

Related Work

This chapter describes the current state of the art of the research related to this work. In the first section we discuss the use of metrics in the context of software engineering. Next, we focus our attention on quality assessment of model transformations. The subsequent section goes into the work related to the automated extraction of metrics. Finally, we indicate how the work presented in this thesis contributes to the current state of the art.

2.1 Software Measurement

Measuring is essential for any engineering activity and for increasing scientific and technical knowledge regarding both the practice of software development and empirical research in software technology [23]. One of the key technologies to measure software artifacts is software metrics. A software metric is an objective measure of some property of a software artifact. Software metrics have been studied extensively in the context of traditional software engineering [31, 12, 22]. Some metrics are measured directly, i.e., by analyzing the static structure of a software artifact, and are relatively easy to extract. Others are more complex and can only be obtained by combining metrics [12].

To assess the quality of a software artifact, an appropriate set of metrics needs to be defined. This set, which is tailored towards a specific formalism, is used to address a set of quality attributes. To this end, the IEEE introduced a standard for software quality metrics methodology [30]. Another frequently used methodology for defining metrics is the Goal-Question-Metric methodology [4]. Alternatively, a more exploratory approach can be applied. In [40, 43], the relations between metrics and quality attributes are examined by means of an empirical study. To prevent that observations are missed, the authors did not restrict their sets of metrics beforehand. Instead, they acquired the metrics by performing brainstorm sessions. In either way, the obtained metric values are used to provide insight into the quality of software artifacts.

2.2 Quality Assessment of Model Transformations

As proposed in [39], the quality of model transformations can be viewed in two different ways. The internal quality of a model transformation is the quality of the transformation
artifact itself, whereas its external quality concerns quality aspects that are observable from the outside [39], i.e., the quality change it induces on a model.

In [32], an indirect way of quality assessment is advocated to measure the external quality of model transformations. This is done by comparing the quality of an input model with the quality of the generated output model. However, when one wants to define and assess the quality of the transformation artifact itself, i.e., its internal quality, direct quality assessment is considered the appropriate approach [38].

In [40], the internal quality of a model transformation is measured by measuring the model transformation itself. To this end, the authors defined a set of metrics for Text-to-Text (T2T) model transformations specified in ASF+SDF [45], a grammarware formalism. Since metrics alone are not enough for assessing quality, they related these metrics to quality attributes by means of an empirical study.

In [42, 51, 29, 43], similar approach have been applied for modelware formalisms ATL [24], QVTO [18, 26] and Xtend [52]. The overlap between the sets of metrics defined for these formalisms is shown in [37]. The authors concluded that even though the formalisms have different characteristics, many metrics are similar. Nevertheless, each of the formalisms also has some specific metrics. These metrics are used to measure aspects of a transformation that are specific for the paradigm and the range of ideas implemented in the respective formalism [37].

2.3 Automated Quality Assessment

Recall that a number of studies have been conducted related to quality assessment of model transformations by means of metrics. Since manual extraction of metrics is error prone, and becomes infeasible as the size of a model transformation grows, tools are usually developed to perform this task in an automated fashion. Currently, automated metrics extraction is performed by tools tailored towards the formalism at hand [40, 42, 29].

For the extraction of metrics defined for model transformations specified in ASF+SDF, a tool has been developed that consists of two decoupled parts, viz., a front-end and a back-end. The front-end, which is implemented in C, reads an ASF+SDF specification and uses the ASF+SDF API to extract the facts that are required to perform the measurements. The extracted facts are stored in a relational database. The back-end of the tool, which is implemented in Java, eventually uses these facts to calculate the desired metric values.

A different approach has been applied for the extraction of metrics defined for model transformations specified in ATL, QVTO and Xtend. However, the tools developed for this task consist of two decoupled parts as well. The front-end of these tools examines the model transformation, which is represented by a language specific model, and creates a metrics model as an intermediate representation. The back-end, which is actually a pretty printer, takes the intermediate representation as input and produces the values of the desired metrics as output. Since the intermediate metrics model is generic, the back-end of these tools only needed to be implemented once.
2.4 Thesis Contribution

Recall that a number of studies have been conducted in which direct quality assessment, as proposed in [38], has been applied to assess the internal quality of model transformations. Currently, this has been done for grammarware formalism ASF+SDF and modelware formalisms ATL, QVTO and Xtend. Answering research question RQ1, as stated in Section 1.2, will increase our understanding of the suitability of this approach in the context of graph-based model transformations. In addition, the obtained observations could eventually be used to propose a methodology which, if adhered to, leads to high-quality model transformations [40].

Recall also that a number of tools have been developed to automate the extraction of metrics. Although these tools all have their language specific part, abstracting from this reveals a considerable amount of overlap in functionality. Answering research questions RQ2 and RQ3, as stated in Section 1.2, may result in a suitable alternative to the current approach of automated metrics extraction, viz., developing tools tailored towards the formalism at hand.
Chapter 3

Viatra2

This chapter presents a number of preliminary concepts related to model transformation in general, and graph transformation in particular. These basic concepts need to be understood, before we introduce the graph-based model transformation language of the Viatra2 framework. As indicated in Section 1.2, this language is used for answering research question RQ1.

3.1 Basic Concepts

Model transformation is considered the heart and soul of MDE [34]. It provides a mechanism for creating target models, based on information contained in existing source models. Figure 3.1 schematically depicts the context of a model transformation. It shows a transformation $t$ from source model $M_S$ to target model $M_T$. Models $M_S$ and $M_T$ both represent an abstract view of a software artifact and conform to their corresponding metamodels $MM_S$ and $MM_T$, respectively, where possibly $MM_S = MM_T$.

Figure 3.1: Context of model transformation
A metamodel defines the concepts, and relations between these concepts, for creating a class of models, e.g., model $M_S$ is an instance of the class of models created by metamodel $MM_S$. Metamodels $MM_S$ and $MM_T$ conform to meta-metamodels $MMM_S$ and $MMM_T$, respectively, where possibly $MMM_S = MMM_T$. The latter two are specified in terms of themselves. Hence, (meta-)metamodels can be considered models as well.

Transformation $t$ is a runtime instance of transformation definition $TD$. Definition $TD$ can also be considered a model that conforms to metamodel $TF$. That is, transformation definition $TD$ is defined in transformation formalism $TF$. Furthermore, formalism $TF$ may also conform to $MMM_S$ or $MMM_T$, e.g., ATL [24] conforms to its meta-metamodel $MMM_S$, where $MMM_S = MMM_T$.

### 3.2 Model Transformation based on Graph Transformation

Literature related to model transformation considers many software artifacts as potential transformation subject, e.g., UML diagrams, Entity-Relation Diagrams (ERDs) and source code. The varied nature of models further invites specialized transformation approaches that are geared to transforming particular kinds of models [8]. One of these approaches is the graph transformation approach [10].

Many models are graph-based, e.g., UML diagrams, and are therefore most naturally represented as graphs. A graph $G = (V,E)$ consists of a set of vertices $V$ and a set of edges $E$ such that each edge in $E$ has a source and a target vertex in $V$, respectively.

In the context of graph transformation, graphs occur at two levels, viz., instance level and type level. An instance graph is a graph that represents a model by modeling entities as vertices and relations between entities as edges. Both vertices and edges may be typed. Furthermore, they may be attributed. A type graph defines the concepts, and relations between these concepts, for creating a class of instance graphs, i.e., a type graph creates a class of instance graphs, as a metamodel creates a class of models.

![Figure 3.2: Context of graph transformation](image)

Graph transformation can serve as a mechanism for specifying and executing model transformations. A graph transformation rule $r : L \rightarrow R$ consists of a pair of graph patterns. Left-hand side $L$ represent the precondition of rule $r$, whereas right-hand side $R$ describes its postcondition. In case $L$ can be matched by subgraph $S$ of the instance graph under transformation, subgraph $S$ is replaced by $R$.

Figure 3.2 schematically depicts the context of a model transformation based on graph transformation. It shows a transformation $t'$ from source graph $IG_S$ to target graph $IG_T$. Graphs
3.3 Viatra2 Framework

$IG_S$ and $IG_T$ are instance graphs and correspond to source- and target models $M_S$ and $M_T$, respectively (see Figure 3.1). Also in the context of graph transformation these graphs need to conform to their corresponding 'metagraphs'. Where models $M_S$ and $M_T$ conformed to metamodels $MM_S$ and $MM_T$, instance graphs $IG_S$ and $IG_T$ are typed over type graphs $TG_S$ and $TG_T$, respectively.

In contrast to metamodels $MM_S$ and $MM_T$, type graphs $TG_S$ and $TG_T$ need to be sub-graphs of type graph $TG$. Apart from $TG_S$ and $TG_T$, type graph $TG$ may also contain additional types, relations and attributes, which are needed for the transformation process only. Furthermore, notice that in the context of graph transformation there is no need for 'meta-metagraphs'.

Model transformations based on graph transformation are defined as a graph transformation system $GTS = (TG, TR)$, which consists of type graph $TG$ and a set of transformation rules $TR$. Starting the transformation with instance graph $IG_S$ typed over $TG_S$, it is also typed over $TG$. During the model transformation process the intermediate instance graphs are typed over $TG$. Hence, the generated instance graph $IG_T$ is also typed over $TG$. In case $IG_T$ is also typed over $TG_T$, it fulfills the requirement of being syntactically correct [9].

3.3 Viatra2 Framework

The Viatra2 framework provides a rule- and pattern-based model transformation language for the manipulation of graph-based models. To this end, it combines Graph Transformation (GT) [10] with Abstract State Machines (ASMs) [5]. The framework, which is available as an Eclipse plug-in [48], primarily aims at designing model transformations to support model-based system development with the help of invisible formal methods [47].

3.4 Viatra2 Model Transformation Language

The Viatra2 model transformation language is composed of two sublanguages, viz., Viatra Textual Metamodeling Language (VTML) and Viatra Textual Command Language (VTCL). The former sublanguage is used for the specification of (meta)models, whereas the latter is used for the specification of model transformations. We start this section with a short introduction of VTML. In the remainder, we introduce the main concepts of VTCL.

3.4.1 Viatra Textual Metamodeling Language

Currently, most metamodeling language, e.g., Eclipse Modeling Framework (EMF) [13], are derived with slight variations from the Meta Object Facility (MOF) Core Specification [17]. However, as stated in [49], the MOF standard fails to support multi-level metamodeling [3]. Therefore, VTML is based on the Visual and Precise Metamodeling (VPM) approach [49], which provides a uniform representation of models and metamodels in a VPM model space. The VPM model space consists of entities and relations, representing concepts and relations between these concepts, respectively. A relation can optionally be defined as an aggregation relation. Furthermore, its multiplicity can be defined. Apart from ordinary relations, VPM supports two special kinds of relations, viz., supertype-of and instance-of relations. The
former represents a *superclass-subclass* relation, whereas the latter represents a *type-instance* relation, i.e., between meta-levels.

Figure 3.3 depicts an example of a metamodel specified in VTML. It shows the specification of a UML class diagram, which corresponds to the left diagram depicted in Figure 3.4. Entity *Classifier* is defined as a super type of entities *PrimitiveDataType* and *Class*. To indicate that a class can have superclasses, relation *parent* is defined from entity *Class* to itself. Furthermore, classes can be associated. This is modeled using relations *src* (source) and *dst* (destination), from entity *Association* to entity *Class*. In addition, classes can contain typed attributes. Therefore, relations *type* and *attrs* (attributes) are defined from *Attribute* to *Classifier* and *Class* to *Attribute*, respectively. The latter relation is defined as an aggregation relation with multiplicity many-to-many.

```vbnet
1entity(UML) {  // definition of UML metamodel
2 entity(Classifier);  // entity named Classifier
3 entity(PrimitiveDataType);
4 supertypeOf(Classifier, PrimitiveDataType); // supertype relation between entities
5 entity(Class);
6 supertypeOf(Class, Class);
7 relation(parent, Class, Class); // relation between entities
8 entity(Association);
9 relation(src, Association, Class);
10 relation(dst, Association, Class);
11 entity(Attribute);
12 relation(type, Attribute, Classifier);
13 relation(attrs, Class, Attribute);
14 multiplicity(attrs, many-to-many); // relation is many-to-many relation
15 isAggregation(attrs, true); // relation is aggregation relation
20}
```

Figure 3.3: Example of VTML file

![Figure 3.3: Example of VTML file](image)

**UML**

- **Classifier**
  - **type**
- **Class**
  - **src**
  - **dst**
  - **attrs**
- **Attribute**
  - **type**

**precondition (lhs)**

- **CP:** *Class*
  - **parent**
  - **attrs**
- **CS:** *Class*
  - **parent**
- **A:** *Attribute*
  - **attrs**

**postcondition (rhs)**

- **CP:** *Class*
  - **parent**
  - **attrs**
- **CS:** *Class*
  - **A:** *Attribute*

Figure 3.4: Specification of UML class diagram and corresponding GT patterns
3.4.2 Viatra Textual Command Language

VTCL consists of several language constructs that together form an expressive language for developing both Model-to-Model (M2M) and Model-to-Code (M2C) transformations. A Viatra2 model transformation is specified in terms of one or more VTCL files, each containing an ASM definition. The ASM is either defined in an explicitly defined namespace or in the default namespace. Other namespaces can be imported for direct accessibility of the definitions that are part of them. The body of an ASM is specified in terms of GT patterns, GT rules, ASM rules and ASM function. Figure 3.5 depicts an example of a VTCL file. It shows the definition of an ASM named UmlTransformation in the explicitly defined namespace.umlTrafo. Furthermore, namespace UML is imported for direct accessibility of the UML metamodel depicted in Figure 3.3.

```vtcl
namespace umlTrafo;       // defined namespace
import UML;               // imported namespace
machine UmlTransformation { // ASM definition
  ...
}
```

Figure 3.5: Example of VTCL file

GT Pattern

The most fundamental part of a Viatra2 model transformation is Graph Transformation (GT) patterns. GT patterns in Viatra2 are close to predicates in Prolog, as they have a name, a parameter list and a body. The body of a GT pattern usually contains multiple pattern elements, viz., entities and relations, that together represent a matching condition. GT patterns can be specified in terms of multiple alternative bodies. In case one of these bodies can be fulfilled, the complete GT pattern can be fulfilled. The parameters of a GT pattern can optionally be statically typed.

Besides pattern elements, GT patterns can contain variable assignments and check conditions. The former construct defines the equality of two locally visible variables, whereas the latter defines a boolean condition that must be true for successful pattern matching. In addition, GT patterns can contain negative patterns, which define negative constraints, i.e., if a negative pattern can be matched, the match of its containing GT pattern fails.

GT patterns are either defined inline or standalone. In the former case they are enclosed by other definitions, i.e., GT rules and GT patterns, whereas in the latter case they can be referred to by multiple definitions. Standalone GT patterns can be called by (other) GT patterns, GT rules and ASM rules.

Pattern calls and alternative bodies can be combined for the definition of a recursive GT pattern, i.e., a GT pattern in which one or more bodies define the halt condition for the recursion, while subsequent bodies contain a recursive call to itself.

GT patterns are either defined shareable or non-shareable. In the former case, two pattern elements can be bound to the same model element, whereas in the latter case the injectivity condition is checked, i.e., pattern elements are uniquely bound. An example of a GT pattern
is depicted in Figure 3.6. It shows the specification of a matching condition, which corresponds to the upper right diagram depicted in Figure 3.4. Variables \( CP \) and \( CS \) represent entities of type \( Class \). To indicate that class \( CP \) is a parent of class \( CS \), a relation is defined from \( CS \) to \( CP \). This relation is identified by variable \( Par \) of type \( Class.parent \). In addition, an entity of type \( Attribute \) is defined, which is represented by variable \( A \). To indicate that class \( CS \) contains attribute \( A \), relation \( Attr \) of type \( Class.attrs \) is defined.

Figure 3.6: Example of GT pattern

```plaintext
1 pattern lhs(CP, CS, A, Par, Attr) -> { // name and parameter list
2   Class(CP); // entity of type Class
3   Class.parent(Par, CS, CP); // relation of type Class.parent
4   Class(CS); // entity of type Class
5   Class.attrs(Attr, CS, A); // relation of type Class.attrs
6   Attribute(A); // entity of type Attribute
7 }
```

GT Rule

GT rules in Viatra2 have a name, a parameter list and a body. The body of a GT rule consists of a precondition and an optional postcondition and/or action part. Both the pre- and postcondition of a GT rule are specified in terms of GT patterns, that are either defined inline or standalone. The action part of a GT rule allows the execution of additional actions, viz., ASM rules. The parameters of a GT rule can be statically typed. Furthermore, they are directed by definition. In contrast to output parameters, input parameters are bound to a particular model element before the GT rule is called.

GT rules are always defined as a standalone part of a Viatra2 model transformation. They can be called by other GT rules and ASM rules. Figure 3.7 depicts an example of a GT rule, which corresponds to the right diagrams depicted in Figure 3.4. It shows the specification of a GT rule that lifts the attributes of subclass \( CS \) to its superclass \( CP \). It has three parameters, viz., \( CP \), \( CS \) and \( A \), which are bound to particular model elements before the GT rule is called (see Section 3.4.2). Notice that its precondition equals the GT pattern depicted in Figure 3.6.

The matchings of the precondition pattern are passed to the postcondition pattern via parameter passing, which acts as an explicit interface between pre- and postcondition. That is, the parameters of the postcondition are already bound before calculating its effects on the model space. Since parameter \( Attr \) does not occur in the body of the postcondition, the relation from class \( CS \) to attribute \( A \) is removed. However, a relation between class \( CP \) and attribute \( A \) is created, because variable \( Attr2 \) does not occur in the parameter list of the postcondition. Eventually, an additional ASM print rule is executed.

ASM Rule

To control the order in which GT rules are executed, Viatra2 includes Abstract State Machine (ASM) rules. ASM rules support the definition of complex control flow, including sequencing, if-then-else structures, iterative execution and universally quantified execution.
3.4 Viatra2 Model Transformation Language

Similar to GT patterns, ASM rules are either defined inline or standalone. In contrast to GT patterns, however, inline ASM rules are structurally different from their standalone counterpart. In fact, standalone ASM rules are specified in terms of an inline ASM rule. Furthermore, they have a name and a parameter list. Similar to GT rules, the parameters of a standalone ASM rule are directed by definition and can be statically typed. Standalone ASM rules can be called by other standalone ASM rules. In addition, they can be called in the action part of GT rules. Figure 3.8 depicts an example of a standalone ASM rule. It shows the specification of an ASM iterate rule that applies the GT rule, as specified in Figure 3.7, as long as possible for the entire model space. Notice that this is where the parameters of the GT rule are bound to model elements.

```java
1rule main() { // name and parameter list
2  iterate // ASM iterate rule
3    choose C1, C2, A apply liftAttributes(C1,C2,A); // ASM choose rule
}
```

**Figure 3.8: Example of standalone ASM rule**

**ASM Function**

In addition to aforementioned language constructs, Viatra2 model transformations can be specified in terms of ASM functions. An ASM function is a dynamic array that can be used for the temporary storage of information during the execution of a model transformation. ASM functions have a name, index elements (arity) and optionally a number of initial values. The index elements of an ASM function can be statically typed. The same holds for its return value.

ASM functions are always defined as a standalone part of a Viatra2 model transformation. They can be called by other ASM functions, ASM rules, GT rules and GT patterns. Figure 3.9 depicts an example of an ASM function. It shows the specification of a dynamic array
with arity two, containing three untyped initial values.

```plaintext
lasmfunction table / 2 {  // name and untyped index elements (arity)
  (1, "color") = "Red";  // initial value
  (7, "color") = "Blue";  // initial value
  (7.1, 6) = 2;          // initial value
}
```

Figure 3.9: Example of ASM function
Chapter 4

Metrics Definition

This chapter describes the work that we performed to answer research question RQ1, i.e., can we apply direct quality assessment, as proposed in [38], to assess the internal quality of model transformations based on graph transformation?

In the first section of this chapter we discuss the applied approach. Next, we indicate the quality attributes that we intend to address. In the subsequent section we describe the metrics that we defined for this purpose. Furthermore, we discuss the relations that we identified between the quality attributes and the defined metrics. Finally, we summarize the insights that we acquired.

4.1 Viatra2 Model Transformation Assessment

To answer research question RQ1, we applied a similar approach as described in Section 2.2. That is, we defined a set of metrics to assess the internal quality of model transformations specified in the Viatra2 model transformation language. Since metrics alone are not enough for assessing quality, we related these metrics to quality attributes.

In [40, 43], the obtained metrics data has been correlated with expert data, to assess whether the defined metrics are appropriate predictors for the chosen quality attributes. In another recent study [29], the authors related the defined metrics to quality attributes based on their expectations and considered the validation of these relations future work.

To come to meaningful conclusions from an empirical study, it is crucial to have a considerable amount of empirical data. Since the user community of Viatra2 is considered rather small, we decided to restrict ourselves to the latter approach as well. That is, we related the defined metrics to quality attributes on the basis of our expectations.

4.2 Quality Attributes

The quality attributes that we intend to address are understandability, modifiability, reusability, modularity, completeness, consistency and conciseness. Most of these quality attributes can be applied to software artifact in general. In [41], their relevance is mentioned for model transformations. For a more detailed description of these quality attributes in the context of model transformation, we refer to their work.
4.3 Viatra2 Model Transformation Metrics

This section describes the metrics we defined for assessing the quality of Viatra2 model transformations. These metrics can be divided into six categories, viz., VTCL file metrics, GT pattern metrics, GT rule metrics, ASM rule metrics, ASM function metrics and dependency metrics. In the remainder of this section, we address each of these categories and elaborate on the metrics belonging to them.

4.3.1 VTCL File Metrics

A Viatra2 model transformation is specified in terms of one or more VTCL files. To provide insight into the size and modularity of a model transformation, we defined the metric *number of VTCL files*. Since VTCL files optionally contain an explicitly defined namespace, we distinguish between VTCL files with and without defined namespace. Furthermore, we propose the metric *number of imported namespaces per VTCL file*.

A VTCL file contains a single ASM definition, which is specified in terms of GT patterns, GT rules, ASM rules and ASM functions. Since these definitions need to be explicitly called, it may be the case that none of them are called. To detect these unused VTCL files, we included the metric *number of unused VTCL files*. An overview of the VTCL file metrics is given in Table 4.1.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Quality Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>VTCL File</td>
<td>Understandability</td>
</tr>
<tr>
<td># VTCL files</td>
<td>-</td>
</tr>
<tr>
<td># VTCL files with defined namespace</td>
<td>-</td>
</tr>
<tr>
<td># VTCL files without defined namespace</td>
<td>-</td>
</tr>
<tr>
<td># Imported namespaces per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># Unused VTCL file</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.1: VTCL file metrics related to quality attributes

4.3.2 GT Pattern Metrics

GT patterns are either defined inline or standalone. A measure for the size of a model transformation is the number of standalone GT patterns. Two orthogonal distinctions can be made. On one hand, GT patterns are either defined shareable or non-shareable. On the other hand, we can distinguish between recursive and non-recursive GT patterns. In addition, we propose distinct metrics for GT patterns with and without parameters.

GT patterns are uniquely identified by their name and number of parameters. It is possible to overload standalone GT patterns, i.e., define standalone GT patterns with the same name but with a different number of parameters. To measure this kind of overloading, we defined the metrics *number of overloaded standalone GT patterns* and *number of standalone GT patterns per standalone GT pattern name*. 

4.3 Viatra2 Model Transformation Metrics

Standalone GT patterns need to be explicitly called. Therefore, it may be the case that there are standalone GT patterns that are never called. To detect these unused standalone GT patterns, we included the metric number of unused standalone GT patterns.

The complexity of a GT pattern is, among others, determined by its number of pattern bodies. We therefore propose the metric number of pattern bodies per GT pattern. The complexity of a GT pattern may also be influenced by its parameters. To provide insight into the use of parameters, we defined the metric number of parameters per GT pattern. Since parameters can optionally be typed, we distinguish between typed and untyped parameters. Parameters may be unused for various reasons. To detect this, we propose the metric number of unused parameters per GT pattern.

A pattern body usually contains multiple pattern elements, viz., entities and relations. In addition, it can contain negative patterns, which are either defined inline or standalone. In the latter case, the complexity of a GT pattern is also implicitly affected by the complexity of its inline negative patterns. We therefore defined the metric number of pattern elements per GT pattern as the total number of pattern elements, including those defined in its inline negative patterns. The metrics number of entities per GT pattern and number of relations per GT pattern are defined similarly.

The use of inline negative patterns results in a hierarchy, i.e., GT patterns alternated with pattern bodies. As a measure for the complexity of a GT pattern, we propose the metrics height per GT pattern and width per GT pattern. In addition, we defined the metric number of negative patterns per GT pattern as its total number of negative patterns.

Besides pattern elements and negative patterns, GT patterns can contain variable assignments and check conditions. These are measured using the metrics number of variable assignments per GT pattern and number of check conditions per GT pattern, respectively. Again, since these language constructs can occur in inline negative patterns, they are defined as the total number of variable assignments and check conditions, including those defined in inline negative patterns. In addition, we measure the number of local variables per GT pattern, i.e., variables that do not occur in the parameter list of a GT pattern.

Recall that a distinction can be made between inline and standalone GT patterns. To provide insight into the use of GT patterns, we propose distinct metrics for all metrics described above, i.e., when we defined a metric per GT pattern, we actually defined it per inline GT pattern, per standalone GT pattern and the combination of both. Notice that a non-standalone GT pattern is either contained by another GT pattern or a GT rule. Considering the former case as an inline GT pattern, may result in measuring certain language constructs multiple times. Since we rather measure language constructs in isolation, we only consider the latter case as an inline GT pattern. An overview of the GT pattern metrics is given in Table 4.2.

### 4.3.3 GT Rule Metrics

In contrast to GT patterns, GT rules are always defined as a standalone part of a Viatra2 model transformation. A measure for the size of a model transformation is therefore the number of GT rules. GT rules consist of a precondition and an optional postcondition and/or action part. The pre- and postcondition of a GT rule are declarative, whereas its action part is imperative.
To provide insight into the level of declarativeness of a GT rule, we propose the metrics number of GT rules with postcondition per VTCL file, number of GT rules with action part per VTCL file, and the number of GT rules with postcondition and action part per VTCL file. In addition, a distinction can be made between GT rules with and without directed parameters.

GT rules are uniquely identified by their name and number of directed parameters. Similar to standalone GT patterns, GT rules can be overloaded. To measure this kind of overloading we defined the metrics number of overloaded GT rules and number of GT rules per GT rule name. GT rules need to be explicitly called. Since it may be the case that they are never called, we included the metric number of unused GT rules per VTCL file.

Both the pre- and postcondition of a GT rule are defined in terms of GT patterns, that are either defined inline or standalone. Since the size and complexity of a GT rule is strongly determined by its inline GT patterns, we defined similar metrics for these inline GT patterns as we did for their standalone counterparts, i.e., number of inline GT patterns per GT rule with a distinction between shareable and non-shareable, and recursive and non-recursive inline GT patterns.

The complexity of a GT rule may also be influenced by its directed parameters, which is measured using the metric number of directed parameters per GT rule. Since a directed parameter is either an input parameter, output parameter or both, we defined dedicated metrics for them. Furthermore, we distinguish between typed and untyped directed parameters. Di-

<table>
<thead>
<tr>
<th>Metric</th>
<th>Quality Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>S. GT P.</td>
<td>Understandability</td>
</tr>
<tr>
<td># Standalone GT patterns per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># Unused standalone GT patterns per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># Overloaded standalone GT patterns</td>
<td>-</td>
</tr>
<tr>
<td># Standalone GT patterns per standalone GT pattern name (overloadings)</td>
<td>-</td>
</tr>
<tr>
<td># GT patterns with parameters per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># GT patterns without parameters per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># Shareable GT patterns per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># Non-shareable GT patterns per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># Recursive GT patterns per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># Non-recursive GT patterns per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># Parameters per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td># Typed parameters per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td># Untyped parameters per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td># Unused parameters per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td># Pattern bodies per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td># Pattern elements per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td># Entities per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td># Relations per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td># Negative patterns per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td># Variable assignments per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td># Check conditions per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td># Local variables per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td>Height per GT pattern</td>
<td>-</td>
</tr>
<tr>
<td>Width per GT pattern</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.2: GT pattern metrics related to quality attributes
rected parameters may be unused for various reasons. To detect this, we defined the metric *number of unused directed parameters per GT rule*.

Recall that GT rules can contain an additional action part that consists of inline ASM rules. In the following subsection we describe the metrics we defined for ASM rules. Since the size and complexity of a GT rule is strongly influenced by this additional action part, all metrics related to inline ASM rules are similarly defined for GT rules. An overview of the GT rule metrics is given in Table 4.3.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Quality Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td># GT rules per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># GT rules with directed parameters per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># GT rules without directed parameters per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># GT rules with postcondition per VTCL file</td>
<td>- - +</td>
</tr>
<tr>
<td># GT rules with action part per VTCL file</td>
<td>- - -</td>
</tr>
<tr>
<td># GT rules with postcondition and action part per VTCL file</td>
<td>- - -</td>
</tr>
<tr>
<td># Unused GT rules per VTCL file</td>
<td>- -</td>
</tr>
<tr>
<td># Directed parameters per GT rule</td>
<td>-</td>
</tr>
<tr>
<td># Directed in-parameters per GT rule</td>
<td>-</td>
</tr>
<tr>
<td># Directed out-parameters per GT rule</td>
<td>-</td>
</tr>
<tr>
<td># Typed directed parameters per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Untyped directed parameters per GT rule</td>
<td>- - -</td>
</tr>
<tr>
<td># Unused directed parameters per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline GT patterns per GT rule</td>
<td>-</td>
</tr>
<tr>
<td># Shareable inline GT patterns per GT rule</td>
<td>- - +</td>
</tr>
<tr>
<td># Non-shareable inline GT patterns per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Recursive inline GT patterns per GT rule</td>
<td>- - +</td>
</tr>
<tr>
<td># Non-recursive inline GT patterns per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline ASM rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline simple ASM rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline simple ASM skip rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline simple ASM fail rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline simple ASM rule invocation rules per GT rule</td>
<td>- - +</td>
</tr>
<tr>
<td># Inline simple ASM update rules per GT rule</td>
<td>- +</td>
</tr>
<tr>
<td># Inline simple ASM print rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline simple ASM log rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline compound ASM sequential rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline compound ASM parallel rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline compound ASM choose rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline compound ASM forall rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline compound ASM iterate rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline compound ASM if rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline compound ASM try rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline compound ASM variable definition rules per GT rule</td>
<td>+ + - +</td>
</tr>
<tr>
<td># Inline model manipulation ASM rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline model manipulation ASM create rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline model manipulation ASM delete rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline model manipulation ASM copy rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline model manipulation ASM move rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Inline model manipulation ASM element update rules per GT rule</td>
<td>- -</td>
</tr>
<tr>
<td># Overloaded GT rules</td>
<td>- -</td>
</tr>
<tr>
<td># GT rules per GT rule name (overloadings)</td>
<td>- -</td>
</tr>
<tr>
<td>Cyclomatic complexity of action part per GT rule</td>
<td>- -</td>
</tr>
</tbody>
</table>

Table 4.3: GT rule metrics related to quality attributes
4.3.4 ASM Rule Metrics

Similar to GT patterns, ASM rules are either defined inline or standalone. A measure for the size of a model transformation is the number of standalone ASM rules. Since inline ASM rules contribute to the size of their enclosing definitions, we did not define dedicated metrics for them. Similar to GT rules, a distinction can be made between ASM rules with and without directed parameters.

Standalone ASM rules are uniquely identified by their name and number of directed parameters. Similar to GT rules and GT patterns, it is possible to overload standalone ASM rules. To measure this kind of overloading we defined the metrics number of overloaded standalone ASM rules and number of standalone ASM rules per standalone ASM rule name. Standalone ASM rules need to be explicitly called. Since it may be the case that they are never called, we included the metric number of unused standalone ASM rules per VTCL file.

The complexity of a standalone ASM rule is, among others, determined by its number of inline ASM rules. In VTCL, different types of inline ASM rules exist, viz., simple ASM rules, compound ASM rules and model manipulation ASM rules. In contrast to simple ASM rules, compound ASM rules can contain multiple inline ASM rules. Consequently, the complexity of a compound ASM rule is strongly influenced by its inline ASM rules. Model manipulation ASM rules are conceptually different from the former two types, and can be used for manipulating a model in a more direct fashion. Since these three types of ASM rules significantly differ from each other, we propose dedicated metrics for them.

Similarly to traditional programming languages, e.g., Java, C++ and C, the Viatra2 model transformation language supports a number of iterative and branching ASM rules. An alternative measure for the complexity of a standalone ASM rule can therefore be the number of linearly independent paths through it. To this end, we defined the metric cyclomatic complexity per standalone ASM rule, which is closely related to McCabe’s cyclomatic complexity [28]. In contrast to traditional programming languages, VTCL supports parallel composition of inline ASM rules. Since our interest lies in transformation complexity, rather than execution complexity, we follow the same approach as proposed in [44], i.e., we handle parallel composition in the same manner as sequential composition.

The complexity of a standalone ASM rule may also be influenced by its directed parameters, which is measured using the metric number of directed parameters per standalone ASM rule. Since a directed parameter is either an input parameters, output parameter or both, we defined dedicated metrics for them. Furthermore, we distinguish between typed and untyped directed parameters. Directed parameters may be unused for various reasons. To detect this, we defined the metric number of unused directed parameters per standalone ASM rule. An overview of the standalone ASM rule metrics is given in Table 4.4.

4.3.5 ASM Function Metrics

Similar to GT rules, ASM functions are always defined as a standalone part of a Viatra2 model transformation. A measure for the size of a model transformation is therefore the number of ASM functions.
### Table 4.4: Standalone ASM rule metrics related to quality attributes

<table>
<thead>
<tr>
<th>Metric</th>
<th>Quality Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Understandability</td>
</tr>
<tr>
<td># Standalone ASM rules per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># Standalone ASM rules with directed parameters per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># Standalone ASM rules without directed parameters per VTCL file</td>
<td>-</td>
</tr>
<tr>
<td># Direct parameters per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Directed in-parameters per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Directed out-parameters per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Typed directed parameters per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># UnTyped directed parameters per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Unused directed parameters per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Inline ASM rules per standalone ASM</td>
<td>-</td>
</tr>
<tr>
<td># Inlined simple ASM rules per standalone ASM</td>
<td>-</td>
</tr>
<tr>
<td># Inlined ASM skip rules per standalone ASM</td>
<td>-</td>
</tr>
<tr>
<td># Inlined ASM fail rules per standalone ASM</td>
<td>-</td>
</tr>
<tr>
<td># Inlined ASM rule invocation rules per standalone ASM</td>
<td>-</td>
</tr>
<tr>
<td># Inlined simple ASM update rules per standalone ASM</td>
<td>-</td>
</tr>
<tr>
<td># Inlined simple ASM print rules per standalone ASM</td>
<td>-</td>
</tr>
<tr>
<td># Inlined simple ASM log rules per standalone ASM</td>
<td>-</td>
</tr>
<tr>
<td># Inlined compound ASM rules per standalone ASM</td>
<td>-</td>
</tr>
<tr>
<td># Inlined compound ASM sequential rules per standalone ASM</td>
<td>-</td>
</tr>
<tr>
<td># Inlined compound ASM parallel rules per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Inlined compound ASM choose rules per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Inlined compound ASM forall rules per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Inlined compound ASM iterate rules per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Inlined compound ASM if rules per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Inlined compound ASM try rules per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Inlined compound ASM variable definition rules per standalone ASM rule</td>
<td>+</td>
</tr>
<tr>
<td># Inlined model manipulation ASM rules per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Inlined model manipulation ASM create rules per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Inlined model manipulation ASM delete rules per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Inlined model manipulation ASM copy rules per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Inlined model manipulation ASM move rules per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Inlined model manipulation ASM update rules per standalone ASM rule</td>
<td>-</td>
</tr>
<tr>
<td># Overloaded standalone ASM rules</td>
<td>-</td>
</tr>
<tr>
<td># Standalone ASM rules per standalone ASM rule name (overloadings)</td>
<td>-</td>
</tr>
<tr>
<td>Cyclomatic complexity per standalone ASM rule</td>
<td>-</td>
</tr>
</tbody>
</table>

ASM functions may contain initial values. To be more specific about the size of an ASM function, we therefore defined the metrics **number of ASM functions with/without initial values per VTCL file**. In addition, we measure the number of initial values per ASM function.

ASM functions are uniquely identified by their name and number of index elements. Similarly to standalone ASM rules, it is possible to overload ASM functions. Furthermore, they need to be explicitly invoked. We therefore defined similar metrics for ASM functions as we did for standalone ASM rules.

The complexity of an ASM function is determined by its number of index elements. Since index elements can optionally be typed, we distinguish between typed and untyped index elements. Furthermore, we distinguish between typed and untyped return values. An overview of the ASM function metrics is given in Table 4.5.
### 4.3.6 Dependency Metrics

Viatra2 model transformations usually contain numerous references. In total we consider twelve categories of dependency. The following enumeration gives an overview of them:

- Standalone GT patterns depending on standalone GT patterns.
- Standalone GT patterns depending on ASM functions.
- GT rules depending on standalone GT patterns.
- GT rules depending on GT rules.
- GT rules depending on standalone ASM rules.
- GT rules depending on ASM functions.
- Standalone ASM rules depending on standalone GT patterns.
- Standalone ASM rules depending on GT rules.
- Standalone ASM rules depending on standalone ASM rules.
- Standalone ASM rules depending on ASM functions.
- ASM functions depending on ASM functions.
- ASM definitions depending on ASM definitions.

To measure these dependencies we defined a number of fan-in and fan-out metrics. Fan-in of a transformation element \( t \) is the number of times \( t \) is called by transformation element \( t' \). Fan-out of \( t \) is the number of times \( t \) calls a transformation element \( t' \). For instance, the metric *number of call from GT rules per GT rule* measures the fan-out of GT rules.

#### Table 4.5: ASM function metrics related to quality attributes

<table>
<thead>
<tr>
<th>Metric</th>
<th>Quality Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td># ASM functions per VTCL file</td>
<td>Understandability</td>
</tr>
<tr>
<td># ASM functions with typed return value per VTCL file</td>
<td>Modifiability</td>
</tr>
<tr>
<td># ASM functions without typed return value per VTCL file</td>
<td>Reusability</td>
</tr>
<tr>
<td># ASM functions with initial values per VTCL file</td>
<td>Completeness</td>
</tr>
<tr>
<td># ASM functions without initial values per VTCL file</td>
<td>Consistency</td>
</tr>
<tr>
<td># Unused ASM functions per VTCL file</td>
<td>Conciseness</td>
</tr>
<tr>
<td># Index elements per ASM function</td>
<td></td>
</tr>
<tr>
<td># Typed index elements per ASM function</td>
<td></td>
</tr>
<tr>
<td># Untyped index elements per ASM function</td>
<td></td>
</tr>
<tr>
<td># Initial values per ASM function</td>
<td></td>
</tr>
<tr>
<td># Overloaded ASM functions</td>
<td></td>
</tr>
<tr>
<td># ASM functions per ASM function name</td>
<td></td>
</tr>
</tbody>
</table>


4.3 Viatra2 Model Transformation Metrics

Notice that standalone GT patterns and ASM functions can be called both in the pre- and postcondition of a GT rule. To provide insight into the use of these patterns and functions, we distinguish between calls in the pre- and postcondition of a GT rule. Furthermore, a distinction can be made between local and non-local calls, i.e., calls within the same ASM definition and those among different ASM definitions.

The dependency of ASM definitions on other ASM definitions is measured using the metrics number of calls to ASM definitions per ASM definition and number of calls from ASM definitions per ASM definition. An overview of the dependency metrics is given in Tables 4.6 and 4.7. Notice that standalone GT patterns and standalone ASM rules are abbreviated to GT patterns and ASM rules, respectively.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Quality Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td># Calls to GT patterns per GT pattern (fan-in)</td>
<td>Understandability</td>
</tr>
<tr>
<td># Local calls to GT patterns per GT pattern (fan-in)</td>
<td>Modifiability</td>
</tr>
<tr>
<td># Non-local calls to GT patterns per GT pattern (fan-in)</td>
<td>Reusability</td>
</tr>
<tr>
<td># Calls from GT rules to GT patterns per GT pattern (fan-in)</td>
<td>Modularity</td>
</tr>
<tr>
<td># Calls from precondition of GT rules to GT patterns per GT pattern (fan-in)</td>
<td>Completeness</td>
</tr>
<tr>
<td># Calls from postcondition of GT rules to GT patterns per GT pattern (fan-in)</td>
<td>Consistency</td>
</tr>
<tr>
<td># Calls from GT patterns to GT patterns per GT pattern (fan-in)</td>
<td>Conciseness</td>
</tr>
<tr>
<td># Calls from ASM rules to GT patterns per GT pattern (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Calls to GT rules per GT rule (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Local calls to GT rules per GT rule (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Non-local calls to GT rules per GT rule (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Calls from ASM rules to GT rules per GT rule (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Calls from GT rules to GT rules per GT rule (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Calls to ASM rules per ASM rule (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Local calls to ASM rules per ASM rule (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Non-local calls to ASM rules per ASM rule (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Calls from GT rules to ASM rules per ASM rule (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Calls to ASM functions per ASM function (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Local calls to ASM functions per ASM function (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Non-local calls to ASM functions per ASM function (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Calls from GT rules to ASM functions per ASM function (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Calls from precondition of GT rules to ASM functions per ASM function (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Calls from postcondition of GT rules to ASM functions per ASM function (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Calls from GT rules to ASM functions per ASM function (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Calls from GT patterns to ASM functions per ASM function (fan-in)</td>
<td></td>
</tr>
<tr>
<td># Calls from GT patterns per GT pattern (fan-out)</td>
<td></td>
</tr>
<tr>
<td># Local calls from GT patterns per GT pattern (fan-out)</td>
<td></td>
</tr>
<tr>
<td># Non-local calls from GT patterns per GT pattern (fan-out)</td>
<td></td>
</tr>
<tr>
<td># Calls from GT patterns to GT patterns per GT pattern (fan-out)</td>
<td></td>
</tr>
<tr>
<td># Calls from GT rules to ASM rules per GT rule (fan-out)</td>
<td></td>
</tr>
<tr>
<td># Local calls from GT rules per GT rule (fan-out)</td>
<td></td>
</tr>
<tr>
<td># Non-local calls from GT rules per GT rule (fan-out)</td>
<td></td>
</tr>
<tr>
<td># Calls from GT rules to ASM rules per GT rule (fan-out)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Dependency metrics related to quality attributes (1)
4.4 Relating Metrics to Quality Attributes

This section discusses the relations that we identified between quality attributes, indicated in Section 4.2, and metrics, as described in section 4.3. These relations are exclusively based on our expectations. To validate these relations, and to assess whether the defined metrics are appropriate predictors for the quality attributes, an empirical study is required. However, as indicated in Section 4.1, we consider the validation of these relations future work.

An overview of the identified relations is given in Tables 4.1 to 4.7. To indicate the effect of a metric value on a particular quality attribute, we use the symbols ‘+’ and ‘-‘. As expected, the former symbol denotes a positive effect on the quality attribute, whereas the latter denotes a negative effect. Notice that the higher the value of a metric is, the more the related quality attributes are affected by it. In the remainder of this section, we address each of the quality attributes indicated in Section 4.2 and elaborate on their identified relations.

4.4.1 Understandability

The amount of effort required to understand a model transformation is addressed by the quality attribute understandability. The understandability of a model transformation is negatively affected by its size [40]. Many of the defined metrics, e.g., number of VTCL files and number of standalone GT patterns per VTCL file, are a measure of the size of a model transformation. We therefore expect a negative relation between these metrics and the quality attribute understandability.
4.4 Relating Metrics to Quality Attributes

A positive fan-out means that a definition is (partly) specified in terms of other definitions. Since this reduces the complexity of a definition, i.e., by information hiding, it may have a positive effect on understandability. However, it may also require to navigate among definitions to understand their meaning. Since we expect that the former aspect has a greater effect than the latter, fan-out metrics are positively related to understandability.

Metrics that measure the number of ASM variable definition rules are positively related to understandability as well. This type of ASM rule provides the possibility to declare an expression once, and refer to it multiple times (using a variable). Notice that this also positively affects the quality attribute conciseness.

4.4.2 Modifiability

The modifiability of a model transformation indicates the extent to which it can be adapted to provide different or additional functionality. Modifiability is closely related to understandability, because the easier a model transformation is to understand, the easier it is to modify. Metrics related to understandability are therefore similarly related to the quality attribute modifiability.

In addition, we expect a negative relation between fan-in metrics and modifiability. A high fan-in means that a definition is often used by other definitions. Since adapting such definition affects the meaning of other definitions as well, this may result in undesired and/or unexpected behavior. Therefore, definitions with high fan-in need to be adapted with great care, which negatively affects modifiability. However, it may also required fewer modifications. Since we expect the the former aspect has a greater effect than the latter, fan-in metrics are negatively related to modifiability.

4.4.3 Reusability

The extent to which (a part of) a model transformation can be reused by other model transformations is addressed by the quality attribute reusability. The reusability of a model transformation is negatively affected by language constructs that are specific, i.e., constructs that are tailored towards a specific purpose.

Shareable GT rules (which is the default) are expected to be less specific than their non-shareable counterparts. We therefore related the former positively to reusability and the latter negatively. Typed language constructs are also more restrictive, which negatively affects reusability.

GT rules optionally contain an action part, which is imperative by definition. The disadvantage of imperative language constructs, however, is that they may have side effects. Therefore, we expect that imperative language constructs are only used when declarative ones do not suffice (or are not provided). Hence, the existence of an action part may indicate that a GT rule is tailored towards a specific purpose. The metric number of GT rules with postcondition per VTCL file is therefore positively related to reusability, whereas the metric number of GT rules with action part per VTCL file and number of GT rules with postcondition and action part per VTCL file are negatively related. Metrics that measure the number of inline ASM rules per GT rule are also negatively related to reusability.

Model manipulation is either carried out using GT patterns or model manipulation ASM rules. GT patterns are declarative, whereas model manipulation ASM rules are imperative.
For aforementioned reason model manipulation ASM rules may negatively affect reusability. ASM functions optionally contain initial values. Since initial values are generally specific, we expect a negative relation between the metrics measuring this construct and the quality attribute reusability.

A high fan-in means that a definition is often used by other definitions. This can be an indicating that the definition is generic, which positively affects reusability. A high fan-out, however, is expected to have a negative effect on reusability. A high fan-out means that a definition depends on a large number of other definitions. Consequently, it cannot be reused in isolation.

4.4.4 Modularity

The modularity of a model transformation addresses the extent to which it is systematically structured. The number of VTCL files is a metric for measuring the modularity of a model transformation. We therefore related it positively to the quality attribute modularity. A VTCL file can have an explicitly defined namespace. In case it does not have one, it is more likely that its functionality is only used within itself. We therefore expect a negative relation between the metrics measuring this construct and the quality attribute modularity.

Fan-in and fan-out metrics measure dependency between definitions. Local dependency, i.e., dependency between transformation elements that are part of the same ASM definition, increases the cohesion of the ASM definition. Non-local dependency increases the coupling among ASM definitions. High cohesion is generally a sign of well-structured design, whereas high coupling negatively affects the quality of design [35]. We therefore expect a positive relation between metrics measuring local dependency and the quality attribute modularity. Metrics measuring non-local dependency are expected to relate negatively to modularity.

4.4.5 Completeness

The extent to which a model transformation conforms to its requirements is addressed by the quality attribute completeness. The use of ASM print and ASM log rules may indicate that a model transformation is not fully developed yet, i.e., these types of rules are often used for debugging purposes during development.

ASM functions optionally contain initial values. Initial values may also be defined for debugging purpose. The metrics number of ASM functions with initial values per VTCL file and number of initial values per ASM function are therefore negatively related to completeness.

4.4.6 Consistency

The consistency of a model transformation addresses the extent to which it contains no conflicting information. In VTCL, definitions are identified by their name and number of parameters (or arity). It is possible to overload these definitions. Since overloaded definition generally do not have the same content, it negatively affects the consistency of a model transformation.

The existence of a definition suggests that it is a required part of the model transformation, i.e., part of its functional requirements. However, it may be the case that a definition is never
used, which contradicts its suggestion of importance. Metrics referring to unused definitions are therefore negatively related to consistency.

4.4.7 Conciseness

The extent to which a model transformation does not contain superfluous elements is addressed by the quality attribute conciseness. From a VTCL file, definitions in other VTCL files can be referred to using their Fully Qualified Name (FQN). However, when their namespaces are imported, they can be referred to using their simple name. Since simple names are generally considerably shorter than FQNs, we expect a positive relation between the metric *number of imported namespaces per VTCL file* and the quality attribute conciseness.

GT patterns can be defined recursively. Since recursion generally increases the conciseness of a definition, we positively related the metrics *number of recursive GT patterns per VTCL file* and *number of recursive inline GT patterns per GT rule* to conciseness.

Definitions that are never used, however, have a negative effect on conciseness, i.e., removing them would increase the conciseness of a model transformation. Metrics referring to unused definitions are therefore negatively related to conciseness.

A high fan-in means that a definition is often used by other definitions. As indicated in Section 4.4.3, this can be an indication that the definition is generic, which positively affects reusability. In addition, it may reduce redundancy. We therefore expect a positive relation between fan-in metrics and the quality attribute conciseness.

4.5 Observations

In this chapter we defined a set of metrics to assess the internal quality of model transformations specified in the Viatra2 model transformation language. Since metrics alone are not enough to assess quality, we related these metrics to quality attributes. Since the user community of Viatra2 was considered rather small, we identified these relations on the basis of our expectations. Although an empirical study is required to validate these relations, we are convinced that direct quality assessment can be applied to assess the internal quality of model transformations based on graph transformation.
Chapter 5

Towards a Generic Approach

In this chapter we aim to answer research question RQ2, i.e., can we realize automated metrics extraction for model transformations using a generic approach?

In the first section we discuss the traditional approach of automated metrics extraction. In the subsequent sections we describe the steps towards an appropriate generic approach. Finally, we summarize the insights that we acquired.

5.1 Traditional Approach

As indicated in Section 2.3, a number of tools have been developed to automate the extraction of metrics [40, 42, 29]. Currently, this task is performed by tools tailored towards the formalism at hand. Figure 5.1 depicts the general architecture of these tools, which consist of two decoupled parts, viz., a front-end and a back-end.

![Figure 5.1: Architecture of tool according to traditional approach](image)

The front-end of these tools examines the model transformation, which is represented by a language specific model, and creates a metrics model as an intermediate representation. The back-end, which is actually a pretty printer, takes the intermediate representation as input and produces the values of the desired metrics as output. Since the intermediate metrics model is generic, the back-end of these tools only need to be implemented once. In this way, the amount of language-specific functionality has been reduced. This reduction also holds for future metrics extraction tools, which positively affects their development time.

However, the extraction of metrics is obviously the main task of a metrics extraction tool. The language specific front-end therefore still contains the largest part of the total amount of functionality. Furthermore, abstracting from language specific details reveals a considerable
amount of overlap in functionality. Since the development of a front-end is a time consuming task, it would be beneficial to further reduce this undesired amount of overlap.

5.2 Intermediate Generic Representation

An alternative approach is the use of an intermediate generic model. That is, an intermediate representation that abstracts from language specific details. We assumed that abstracting from these details would reveal that model transformation formalisms provide similar language concepts. An intermediate representation would enable the generalization of automated metrics extraction. The general architecture of a tool using this approach is depicted in figure 5.2.

In comparison with the architecture depicted in Figure 5.1, the process of metrics extraction is moved from the language specific front-end to the generic back-end. Consequently, all functionality related to metrics extraction only needs to be implemented once. The only task that still remains to be done, is the transformation of the language specific model to the intermediate generic model. This model transformation is expected to be rather straightforward, i.e., only the constructs one wants to measure need to be part of the output model. Hence, the level of abstraction is raised.

The question that remains is what the intermediate representation should look like. Answering this question requires focusing on a representative set of model transformation formalisms. To this end, we decided to consider Viatra2 [47] as well as the modelware formalisms ATL [24], QVTO [18, 26] and Xtend [52]. When studying the language constructs provided by these formalisms, a considerable amount of overlap is observable. For instance, all considered formalisms provide transformation functions, which take model elements as input and produce model elements as output. Furthermore, they all have support for structuring model transformations by packaging transformation functions in modules. Imperative language constructs are also supported by all considered formalisms. However, each of the formalisms also has some concepts that are specific for the paradigm and the range of ideas implemented in the respective formalism [37]. In Viatra2, it is possible to define standalone matching conditions. ATL provides constructs for explicit trace resolution. In QVTO, it is possible to define intermediate classes and properties. Xtend has support for aspect-oriented programming.

A possible approach would be to design the generic model in such a way that it only preserves the language constructs the considered formalisms have in common. It is unlikely, however, that this approach is appropriate for extracting the complete sets of defined metrics. Another approach would be to preserve all language constructs provided by the formalisms. However,
even if it would be possible to design the generic model in this way, we do not have any guarantee that it will also be appropriate for other/future model transformation formalisms. Hence, we can conclude that approaches using an intermediate representation are too restrictive.

5.3 No Intermediate Representation

To come up with a less restrictive approach, we need to examine the possibilities of leaving out the intermediate generic representation. The process of metrics extraction, however, must still be realized in a generic fashion. We therefore consider the approach of directly giving the language specific model as input to a generic back-end. The general architecture of a tool using this approach is depicted in figure 5.3.

![Figure 5.3: Architecture of tool without intermediate representation](image)

Similar to the architecture depicted in Figure 5.2, the process of metrics extraction is part of the generic back-end. Hence, it only needs to be implemented once. In contrast to that architecture, however, the intermediate representation is left out, i.e., the generic back-end directly takes the language specific model as input. The advantage of this approach is that we do not lose any language specific details. Furthermore, no additional model transformation is required. The problem of this approach, however, is that the metamodel corresponding to the language specific model is different for each model transformation formalism. Consequently, it is unclear how the metrics extractor, which is part of the generic back-end, should interpret the language specific model that is given as input.

5.4 Semantic Enrichment

A possible solution to the problem as stated in the previous section is semantically enriching the language specific metamodel. That is, an enrichment that enables the interpretation of the language specific model with respect to the extraction of metrics. Notice that metrics are generally defined on different levels of hierarchy. A metric defined on a certain level is an objective measure of a property on that particular level. By explicitly indicating which metaclasses must be interpreted as levels of hierarchy, it becomes clear which structural properties of the language specific metamodel, e.g., metaclasses, attributes and references, must be measured (and possibly aggregated). Similarly, we can explicitly indicate the existence of dependencies, tree structures, decision points and other properties of interest. Figure 5.4 depicts the general architecture of a tool using this approach. Again, its architecture consists of two decoupled parts, viz., a front-end and a back-end.
In contrast to the architectures depicted in Figures 5.2 and 5.3, the task of the front-end is restricted to semantically enriching the language specific metamodel. The back-end takes the enriched metamodel, and its corresponding model, as input and produces the values of the desired metrics as output. The advantage of this approach is that the amount of overlap in functionality is maximally reduced, i.e., all functionality related to metrics extraction is moved to the generic back-end. In fact, the amount of functionality in the language specific front-end is reduced to none. An additional advantage is that this approach is not restricted to model transformation formalisms, i.e., an arbitrary model can be interpreted by the back-end, as long as its corresponding metamodel is semantically enriched.

The question that remains is how the language specific metamodel can be semantically enriched. The answer to this question is twofold. First, semantics need to be represented. For this purpose, ordinary metaclasses can be used (see Chapter 6). Second, these semantic metaclasses have to be associated with the language specific metamodel. To this end, the Meta Object Facility (MOF) Core specification [17] includes a concept called tags. A tag provides the ability to annotate model elements with additional information. It represents a single piece of information that can be associated with multiple model element. Hence, by associating a tag to both a language specific model element and a semantic metaclass, we are able to semantically enrich the language specific metamodel. This idea is depicted in figure 5.5.

An alternative approach for the association of semantic metaclasses with the language specific metamodel is the use of super type relations. By setting a semantic metaclass as a super type of a language specific metaclass, we are able to semantically enrich the language specific metamodel. However, this approach is restricted to metaclasses. Model elements different from metaclasses, e.g., references, attributes and literals, cannot be enriched. Furthermore, in comparison with tags, a super type relation does not provide the ability of adding additional information. Although in some cases these restrictions can be overcome, tags provide considerably more flexibility.
5.5 Observations

In this chapter we described two alternative approaches to the traditional approach of automated metrics extraction, i.e., developing a tool tailored towards the formalism at hand.

The first approach that we considered was the use of an intermediate generic model, i.e., an intermediate representation that abstract from language specific details. We assumed that abstracting from these details would reveal that model transformation formalisms provide similar language constructs. However, after studying a representative set of model transformation formalism, we observed that each of the formalisms also has some language specific concepts. Consequently, it was unclear what the intermediate representation should look like. Only preserving the language constructs the considered formalism have in common, would result in incomplete sets of metrics. Preserving all language constructs did not guarantee that the intermediate representation will be suited for other/future model transformation formalism as well. Therefore, we concluded that the use of an intermediate generic model is too restrictive.

An alternative approach that we considered was the semantic enrichment of a metamodel. That is, an enrichment that enables the interpretation of the language specific model with respect to the extraction of metrics. We observed that the advantage of this approach is that all functionality related to metrics extraction can be implemented language independently. The approach is therefore suited for the extraction of metrics from an arbitrary model (transformation). In the following chapter we elaborate on this approach.
Chapter 6

Semantic Enrichment

This chapter describes how semantic enrichment of a metamodel, as proposed in Chapter 5, enables the interpretation of its corresponding model with respect to the extraction of metrics. In the first section we introduce the semantic metamodel. In the subsequent sections we address each of its semantic metaclasses. Finally, we summarize the insights that we acquired. The ideas presented in this chapter are implemented in a metrics extraction tool. This tool is described in Chapter 7.

6.1 Semantic Metamodel

As indicated in Section 5.4, semantics can be represented using ordinary metaclasses. In total we defined seven metaclasses, each representing particular semantics. Figure 6.1 gives an overview of the metaclasses that are part of the semantic metamodel.

![Figure 6.1: Semantic metamodel](image)

These metaclasses are defined for enabling the extraction of the Viatra2 metrics defined in Chapter 4. However, they are suited for the extraction of metrics from an arbitrary model (transformation). In the remainder of this chapter, we elaborate on these metaclasses. To clarify their semantics, we describe their application using the Viatra2 metamodel (see Figures 7.5 and 7.6). This language specific metamodel is elaborated on in Chapter 7.

Each semantic class has an associated set of metrics. That is, the semantics represented by
a semantic class enables the extraction of a particular category of metrics. These associated sets of metrics are mentioned as well.

6.2 Construct

Metrics are generally defined on different levels of hierarchy. A metric defined on a certain level is an objective measure of a property on that particular level. To provide the ability of explicitly indicating a level of hierarchy, we defined the semantic class *Construct*.

6.2.1 Construct Semantics

A construct represents a language construct on a particular level of hierarchy. That is, a language construct having properties that can be measured, e.g., pattern, rule and module. Since properties must always be measured on a certain level of hierarchy, constructs are the primary means for enabling the interpretation of language specific models with respect to the extraction of metrics. That is, the semantic classes described in subsequent sections must be used in combination with constructs, i.e., otherwise their semantics are undefined.

A construct can be associated with an arbitrary language specific class. Furthermore, multiple classes in a language specific metamodel can be associated with a construct. In case an associated class has subclasses, also its subclasses are considered constructs. These subconstructs are considered specializations of their corresponding construct.

6.2.2 Construct Metrics

Figure 6.2 depicts the context of a construct. It shows that language specific class \( C \) is associated with semantic class *Construct*. This association explicitly indicates that class \( C \) must be interpreted as a language construct on a particular level of hierarchy.

![Figure 6.2: Context of construct](image)

The semantic enrichment of class \( C \) enables the extraction of a particular category of metrics. Namely, metrics related to the structural properties of a construct, viz., class \( C \). For each class in the containment hierarchy rooted at class \( C \), viz., classes \( C \) and \( D \), we measure for each of
6.2 Construct

its (inherited) reference, viz., references \textit{ref3} and \textit{ref4}, the number of referred class instances, viz., instances of classes \( D \) and \( E \).

Three distinctions can be made. In case a class contains an attribute, we can distinguish among all possible attribute values, e.g., we can distinguish between instances of class \( E \) with attribute value \textit{true} and \textit{false}. Notice that we can only consider attributes with a finite number of values, viz., booleans and enumerated types. Considering other attributes would result in an infinite number of metrics. Furthermore, a distinction can be made based on the references of a class. Since class \( C \) refers to class \( D \), we can distinguish between instances of class \( C \) with and without instances of class \( D \). The former category can be further subdivided into instances with a single instance of class \( D \) and with multiple instances of class \( D \). Notice that the same distinctions can be made for instances of class \( E \). Finally, a distinction can be made based on the subclasses of a class, e.g., we can distinguish between instances of classes \( E \) and \( F \).

Recall that a construct can be associated with an arbitrary language specific class. In case the associated class is different from the root of the metamodel, it can have multiple instances. To provide insights on the level of a software artifact, e.g., model transformation, we aggregate all measurements on the level of its constructs. Hence, we aggregate all measurements on the level of class \( C \). To increase the insights even further, the containment of a construct can be taken into account. That is, a distinction can be made between instances of class \( C \) contained by an instances of class \( A \) and an instance of class \( B \). The former category can be further subdivided into instances referred by reference \textit{ref1} and reference \textit{ref2}.

6.2.3 Construct Application

Figure 6.3 depicts a simplified fraction of the Viatra2 metamodel. It shows that an ASM definition can contain multiple GT rules. A GT rule is expressed in terms of a pre- and postcondition. Furthermore, both conditions optionally contain an inline GT pattern, which is either defined shareable or non-shareable.

![Figure 6.3: Application of construct](image)

As indicated in Section 4.3.3, various metrics are defined on the level of a GT rule. To extract these metrics, we need to explicitly indicate that a GT rule must be interpreted as a language construct on a particular level of hierarchy. Therefore, we associate language specific class \textit{GtRuleDef} with semantic class \textit{Construct}. To provide insights on the level of a Viatra2 model transformation, measurements on the level of class \textit{GtRuleDef} are aggregated. The following enumeration gives an overview of the associated set of metrics:
6.3 Id

Language constructs are often identified using an attribute, e.g., name. In case the identifying attribute value is not required to be unique, multiple instances may exist with the same value, e.g., overloading. To provide the ability of explicitly indicating the identifying attribute, we defined the semantic class $Id$.

6.3.1 Id Semantics

An id represents the identifying attribute of a construct. Ids must therefore always be used in combination with constructs, i.e., otherwise its semantics is undefined. An id can be associated with an arbitrary (inherited) attribute of a construct. Recall that a construct can have specializing sub-constructs, viz., subclasses of the represented language specific class. Associating an id with the identifying attribute of a construct, implicitly indicates the identifying attribute of its sub-constructs.

6.3.2 Id Metrics

Figure 6.4 depicts the context of an id. It shows, among others, that language specific class B and its attribute $name$ are associated with semantic classes $Construct$ and $Id$, respectively. These associations explicitly indicate that class $B$ is identified by attribute $name$.

The semantic enrichment of class $B$ and attribute $name$ enables the extraction of a particular category of metrics. Namely, metrics related to (non-)uniquely identified constructs. Multiple instances of class $B$ may exist with the same identifying value of attribute $name$. Therefore, we measure the number of (non-)uniquely identified instances of class $B$. Notice, however, that these measurements only provide meaningful insights, if they are measured per another language construct, e.g., the number of (non-)uniquely identified functions per module. Therefore, these measurements are only generated if an identified construct, i.e., a construct having an identifying attribute, occurs in the containment hierarchy of a(another) construct. That is, we measure the number of (non-)uniquely identified instances of class $B$ per instance of class $A$. Notice that the containing construct can possibly be the identified construct itself, viz., class $A$ equals class $B$. In addition, we measure the number of instances of class $B$ per identifying value of attribute $name$. 
Figure 6.4: Context of id

6.3.3 Id Application

Figure 6.5 depicts a simplified fraction of the Viatra2 metamodel. It shows that a VTCL file can indirectly contain multiple GT rules. A GT rule is identified by its inherited attribute name.

As indicated in Section 4.3.3, GT rules can be overloaded. To measure this form of inconsistency, we need to explicitly indicate that attribute name must be interpreted as the identifying attribute of a GT rule. Therefore, we associate language specific class GtRuleDef and its attribute name with semantic classes Construct and Id, respectively. Furthermore, language specific class VTCLFile is associated with semantic class Construct as well. The following enumeration gives an overview of the associated set of metrics:

- # GtRuleDefs per GtRuleDef name
- # (Non-)uniquely identified GtRuleDefs per VTCLFile

6.4 Counter

Software artifacts may contain properties that are not directly related to a single language construct, e.g., McCabe’s cyclomatic complexity [28]. To provide the ability of counting a collection of language constructs, we defined the semantic class Counter.

6.4.1 Counter Semantics

A counter represents a language construct that can be counter in combination with other language constructs to measure a certain property, e.g., a construct representing a decision point to measure the cyclomatic complexity. Recall that a property needs to be measured on a certain level of hierarchy. Counters must therefore always be used in combination with constructs, i.e., otherwise its semantics is undefined.
A counter can be associated with an arbitrary language specific model element, occurring in the containment hierarchy of a construct. Furthermore, multiple model elements in this hierarchy can be associated with a counter. In case these model elements together represent a single property, their connection must be explicitly indicated. As described in Section 5.4, tags can be provided with additional information. By giving the concerning tags an identical name, we are able to indicate this connection.

Notice that a counter can occur in the containment hierarchy of multiple constructs. Consequently, it can represent a property of multiple constructs as well.

### 6.4.2 Counter Metrics

Figure 6.6 depicts the context of a counter. It shows that language specific class $A$ is associated with semantic class $Construct$. Furthermore, language specific classes $B$ and $C$ are associated with semantic class $Counter$. These associations explicitly indicate that classes $B$ and $C$ must be interpreted as properties of class $A$. In addition, the concerning tags are given the identical name $Counter$. This name explicitly indicates that classes $B$ and $C$ together represent a single property named $Counter$. 
The semantic enrichment of classes \( A, B \) and \( C \) enables the extraction of a particular category of metrics. Namely, metrics related to properties that are not directly related to a single language construct. That is, for each instance of class \( A \), we measure the sum of the number of instances of classes \( B \) and \( C \).

Apart from classes, a counter can also be associated with other model elements, viz., references and literals. In the former case we measure the number of referred class instances. In the latter case the same is done for the occurrences of a particular literal.

### 6.4.3 Counter Application

Figure 6.7 depicts a simplified fraction of the Viatra2 metamodel. It shows that a standalone ASM rule (\( AsmRuleDef \)) is specified in terms of an inline ASM rule (\( AsmRule \)). In case the inline ASM rule is a compound ASM rule, it can contain multiple inline ASM rules itself.

As indicated in Section 4.3.4, some types of inline ASM rules represent a decision point, viz., ASM iterate rule, ASM if rule and ASM try rule. To measure the number of decision points, we need to explicitly indicate that these inline ASM rules must be interpreted as such. Therefore, we associate language specific class \( AsmRuleDef \) with semantic class \( Construct \). Furthermore, language specific classes \( IterateRule, IfRule \) and \( TryRule \) are associated with semantic class \( Counter \). To indicate that these classes together represent a single property, we give the concerning tags the identical name \( DecisionPoint \). The associated metric is:

- \( \# \) DecisionPoints per \( AsmRuleDef \)

### 6.4.4 Counter Specialization: Cyclomatic Complexity

Recall that McCabe’s Cyclomatic Complexity (MCC) is a property that is not directly related to a single language construct. It measures the number of linearly independent paths, which is closely related to the number of decision points.
In the previous subsection we described how the number of decision points can be measured by means of counters. However, MCC is specified as the number of decision points+1. To provide the ability of measuring MCC, we defined the semantic class \textit{CyclomaticComplexity}. A ‘cyclomatic complexity’ is similarly defined as a counter, i.e., it is a specialization that increments (combined) measurements by 1. Hence, by replacing semantic class \textit{Counter} by semantic class \textit{CyclomaticComplexity} (see Figure 6.7), we are able to measure the cyclomatic complexity of a standalone ASM rule.

### 6.5 Dependency Reference

Software artifacts are often expressed in terms of language constructs that depend on each other. To provide the ability of measuring these dependencies, we defined the semantic class \textit{DependencyReference}.

#### 6.5.1 Dependency Reference Semantics

A dependency reference represents a dependency between two, possible the same, constructs. Dependency references must therefore always be used in combination with constructs, i.e., otherwise its semantics is undefined. A dependency reference can be associated with an arbitrary reference of a class, occurring in the containment hierarchy of a construct. Furthermore, multiple references in this hierarchy can be associated with a dependency reference. Notice that a dependency reference can occur in the containment hierarchy of multiple constructs. Consequently, it can represent a dependency of multiple constructs as well. Similarly, it can represent a dependency on multiple constructs.

#### 6.5.2 Dependency Reference Metrics

Figure 6.8 depicts the context of a dependency reference. It shows, among others, that language specific classes \(B\) and \(C\) are associated with semantic class \textit{Construct}. Furthermore, language specific reference \textit{dependencies} is associated with semantic class \textit{DependencyReference}. These associations explicitly indicate that class \(B\) depends on class \(C\).

The semantic enrichment of reference \textit{dependencies} and classes \(B\) and \(C\) enables the extraction of a particular category of metrics. Namely, fan-in and fan-out metrics. For each instance of class \(A\), we measure its number of dependencies on instances of class \(B\), i.e., fan-out of class \(A\). Contrary, for each instance of class \(B\), we measure the number of depending instances of class \(A\), i.e., fan-in of class \(B\).

In addition, we measure the number of (un)called instances of class \(B\). Notice, however, that these measurements only provide meaningful insights, if they are measured per another language construct, e.g., the number of (un)called functions per module. Therefore, these measurements are only generated if a dependable construct, i.e., a construct on which a construct depends, occurs in the containment hierarchy of a(nother) construct. That is, we measure the number of (un)called instances of class \(C\) per instance of class \(A\).

Furthermore, a distinction can be made between local and non-local dependencies, i.e., dependencies within the same instance of class \(A\) or between different instances. In case classes \(B\) and \(C\) are not contained by a similar construct, dependencies are non-local by definition.
6.5.3 Dependency Reference Application

Figure 6.9 depicts a simplified fraction of the Viatra2 metamodel. It shows that a VTCL file can indirectly contain multiple standalone GT patterns. A GT pattern is expressed in terms of pattern bodies, which optionally call a standalone GT pattern.

As indicated in Section 4.3.6, various fan-in and fan-out metrics are defined that measure dependencies between standalone GT patterns. To generate these metrics, we associate language specific classes VTCLFile and GtPatternDef with semantic class Construct. Furthermore, language specific reference gtPatternDefCalls is associated with semantic class DependencyReference.
The following enumeration gives an overview of the associated set of metrics:

- # Calls from GT patterns to GT patterns per GT pattern (fan-out)
- # (Non-)local calls from GT patterns to GT patterns per GT pattern (fan-out)
- # Calls to GT patterns from GT patterns per GT pattern (fan-in)
- # (Non-)local calls to GT patterns from GT patterns per GT pattern (fan-in)
- # (Un)called GT patterns per VTCL file

### 6.6 Tree Reference

Software artifacts are sometimes expressed in terms of language constructs that can form tree structures, e.g., inheritance trees. To provide the ability of measuring these structures, we defined the semantic class `TreeReference`.

#### 6.6.1 Tree Reference Semantics

A tree reference represents a reference connecting two, possible the same, language constructs that result in tree structures. Recall that a property needs to be measured on a certain level of hierarchy. Tree references must therefore always be used in combination with constructs, i.e., otherwise its semantics is undefined.

A tree reference can be associated with an arbitrary reference of a class, occurring in the containment hierarchy of a construct. Furthermore, multiple references in this hierarchy can be associated with a tree reference. In case these references together represent a single tree structure, their connection must be explicitly indicated. By giving the concerning tags an identical name, we are able to indicate this connection.

Notice that a tree reference can occur in the containment hierarchy of multiple constructs. Consequently, it can represent a tree structure of multiple constructs as well.

#### 6.6.2 Tree Reference Metrics

Figure 6.10 depicts the context of a tree reference. It shows that language specific class `A` is associated with semantic class `Construct`. Furthermore, language specific references `ref1` and `ref2` are associated with semantic class `TreeReference`. These associations explicitly indicate that classes `B` and `C` must be interpreted as nodes in a tree structure. In addition, the concerning tags are given the identical name `Tree`. This name explicitly indicates that classes `B` and `C` together represent a single tree structure named `Tree`.

The semantic enrichment of class `A` and references `ref1` and `ref2` enables the extraction of a particular category of metrics. Namely, metrics related to the complexity of tree structures. For each instance of class `A`, we measure the maximum width and height of the tree structures formed by the instances of classes `B` and `C`. Since an instance of class `A` can contain multiple tree structures, we also measure the number of tree structures per instance of class `A`.


6.6 Tree Reference

6.6.3 Tree Reference Application

Figure 6.11 depicts a simplified fraction of the Viatra2 metamodel. It shows that a GT pattern is expressed in terms of pattern bodies. A pattern body optionally contains an inline negative pattern, i.e., a \textit{neg}-keyword followed by a GT pattern.

As indicated in Section 4.3.2, the use of inline negative patterns results in tree structures, i.e., GT patterns alternated with pattern bodies. To measure these structures, we need to explicitly indicate that the concerning references must be interpreted as tree references. Therefore, we associate language specific references \textit{patternBodies}, \textit{gtPatternDef} and \textit{negativePatterns} with semantic class \textit{TreeReference}. Furthermore, language specific class \textit{GtPatternDef} is associated with semantic class \textit{Construct}. To indicate that these three classes together represent a single tree structure, we give the concerning tags the identical name \textit{Tree}. The following enumerations gives an overview of the associated set of metrics:

- # tree structures per GtPatternDef
- Maximum tree width per GtPatternDef
- Maximum tree height per GtPatternDef

6.6.4 Tree Reference Specialization: Dummy Tree Reference

In the previous subsection we described how the complexity of tree structures can be measured by means of tree references. However, in some cases a node should not contribute to the complexity of its tree structure. For this purpose, we defined the semantic class \textit{DummyTreeReference}. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{context_of_tree_reference.png}
\caption{Context of tree reference}
\end{figure}
A dummy tree reference is similarly defined as a tree reference. However, in contrast to ordinary tree references, the instances referred by a dummy tree reference do not contribute to the width and height of the formed tree structures. Dummy tree references must always be combined with tree references, i.e., otherwise its semantics is undefined. Furthermore, their connection with ordinary tree references must be explicitly indicated. By giving the concerning tags an identical name, we are able to indicate this connection.

As indicated in Section 4.3.2, the use of inline negative patterns results in tree structures, i.e., GT patterns alternated with pattern bodies. However, neg-keyword should not contribute to the width and height of the formed tree structures. By associating reference negativePatterns with semantic class DummyTreeReference instead of TreeReference (see Figure 6.11), we are able to explicitly indicate this.
Chapter 7

Automated Metrics Extraction

This chapter presents the work that we performed related to automated metrics extraction. In the first section we describe the metrics extraction tool that implements the ideas presented in Chapter 6. In the subsequent section we go into a separate tool for the generation of Viatra2 models. These language specific models, and their corresponding metamodel, are used for the validation of the approach. This validation is presented in Chapter 8.

7.1 Metrics Extractor

The purpose of the metrics extraction tool is to enable automated metrics extraction according to the ideas presented in Chapter 6. In contract to tools implemented according to the traditional approach (see Section 5.1), the tool that we developed is completely generic. That is, it is suited for the extraction of metrics defined for an arbitrary model transformation formalism. Figure 7.1 depicts its general structure.

![Figure 7.1: Architecture of metrics extractor](image)

The input of the tool consists of an arbitrary model and its corresponding metamodel, representing a model transformation and its defining formalism, respectively. Since the structure of the metamodel is different for each formalism, the metamodel is required to be semantically enriched (see Section 6). That is, without this semantic enrichment it is unclear how the tool should interpret the language specific model.

To examine the structure of the model transformation and its defining formalism, their representing (meta)models are parsed. The resulting in-memory representation is given as input to the metrics extractor.

The process of metrics extraction consists of two separate phases. In the first phase, the metamodel is examined to determine the set of metrics that has to be generated. To this
end, the metamodel is traversed and interpreted according to the semantic enrichment of its model elements. In the second phase, the corresponding model is examined to obtain the desired metric values. To this end, the model is traversed and the encountered instances are measured according to the properties of their corresponding model elements. The output of the tool is a metrics model containing the extracted metric values. Appendix A presents how the metrics extraction tool can be used for the automated extraction of metrics.

7.1.1 Aggregation Techniques

As indicated in Section 6.2, measurements on the level of a construct are aggregated to provide insights on the level of a software artifact, e.g., model transformation. To this end, we implemented nine different aggregation techniques. In addition to a number of traditional aggregation techniques, viz., mean, minimum, maximum, median and standard deviation, we implemented four economic aggregation techniques, viz., Gini index [16], Theil index [36], Kolm Index [25] and Atkinson index [2]. In [50], the results are presented of a comparative study between latter aggregation techniques in the context of software measurement.

7.1.2 Metrics Metamodel

The metric values generated by the metrics extraction tool need to be stored in a certain format. To this end, the Object Management Group (OMG) [19] introduced the Software Metrics Metamodel Specification (SMM) [20]. This specification defines an interchange format between metrics extraction tools. However, the metrics model that we intend to use does not only serve as an interchange format. In fact, its main purpose is to facilitate the examination of the obtained metric values.

Notice that the number of metrics generated by the metrics extraction tool can be overwhelming, e.g., based on the semantic enrichment described in Section 8.2, more than 2000 metrics are generated. To facilitate the examination of these metrics, the metrics metamodel should be designed in such a way that it allows users to examine the generated metric values on different levels of details. Therefore, we decided to design our own metrics metamodel.

In contrast to the SMM specification, the metrics metamodel that we designed also supports the economic aggregation techniques as indicated in the previous section. Figure 7.2 depicts its structure.

As indicated in Section 6.2, metrics can be generated on different levels of hierarchy. A language construct on a particular level of hierarchy is represented by a construct. Therefore, a metrics model can contain multiple constructs. A construct is identified by a name, viz., the name of the represented language specific class. Furthermore, it can contain multiple metrics, measuring properties on the level of the respective construct. A metric is identified by an name and is either a simple metric or an aggregated metric. A metric can contain multiple nested metrics. These nested metrics further specify their containing metric, e.g., a distinction among all possible attribute values.

Recall that a construct can have specializing sub-constructs, viz., subclasses of the represented language specific class. Therefore, a construct in the metrics model can contain multiple sub-constructs. Also recall that when measurements on the level of a construct are aggregated, distinctions can be made based on its containment. These distinctions result in subsets. The metrics belonging to these subsets are contained by a separate construct in the metrics model.
7.1 Metrics Extractor

Figure 7.2: Metrics metamodel

Similar to nested metrics, these sub-constructs and subsets further specify their containing construct. Figure 7.3 depicts an example of a metrics model. It shows a model containing a single construct named $GtPatternDef$. This construct contains four metrics, from which the latter is further specified using two nested metrics, viz., a distinction between typed and untyped pattern body elements.

Figure 7.3: Example of metrics model

7.1.3 Implementation Techniques

To implement the metrics extraction tool according to the ideas presented in Chapter 6, we need to determine suitable techniques. To this end, we decided to use the concepts provided by the Eclipse Modeling Framework (EMF) [13]. EMF is an implementation of the Meta-Object Facility (MOF) Core Specification [17]. It is a modeling framework and code generator facility for building tools based on a structural data model, viz., the Ecore metamodel. The input of the metrics extraction tool is based on this data model. That is, the language specific metamodel and its corresponding model are represented by EMF models. The functionality of the metrics extraction tool is implemented in Java.
EMF Annotations

As indicated in Section 5.4, MOF tags provide the ability to annotate model elements with additional information. The EMF counterpart of the MOF tag is called an annotation. Annotation can be attached to any model element of an EMF metamodel. Furthermore, they can be identified by a name and optionally contain references to other model elements. In our case, these other elements are the semantic classes depicted in Figure 6.1.

Dynamic EMF

The input of the metrics extraction tool consists of an arbitrary model and its corresponding metamodel. Since the structure of the metamodel is different for each formalism, the input of the tool must be handled dynamically. To this end, we make use of dynamic EMF [7]. Dynamic EMF provides the ability to parse the (meta)models that are given as input. It created an in-memory representation, which can be accessed using the EMF reflective API.

7.2 Viatra2 Model Generator

As indicated in Chapter 3, Viatra2 model transformations are specified in terms of VTCL files. Since the metrics extraction tool is developed to handle EMF (meta)models, VTCL files need to be transformed into EMF models. To this end, we developed a Viatra2 model generation tool. Figure 7.4 depicts its general structure.

![Figure 7.4: Architecture of Viatra2 model generator](image)

The input of the tool consists of one or more VTCL files, representing a Viatra2 model transformation. To examine the structure of this model transformation, the representing VTCL files are parsed. The resulting in-memory representation is given as input to a (higher-order) model transformation. This model transformation outputs a Viatra2 model conforming to the Viatra2 metamodel (see Section 7.2.1). Appendix B presents how the Viatra2 model generation tool can be used for the generation of Viatra2 models.

7.2.1 Viatra2 Metamodel

The output of the Viatra2 model generation tool is a Viatra2 model. Figure 7.5 depicts the general structure of its corresponding metamodel. As indicated in Section 4.3.4, different types of inline ASM rules exist, viz., simple ASM rules, compound ASM rules and model manipulation ASM rules. To prevent cluttering, the classes representing these types of inline ASM rules are separately depicted in Figure 7.6. Notice that the Viatra2 metamodel abstract from details that are not required for the extraction of the metrics defined in Section 4.3. It is solely designed for the generation of
models that serve as input for the metrics extraction tool described in Section 7.1. Hence, the model transformation depicted in Figure 7.4 is a vertical transformation that abstracts from unrequired details.

7.2.2 Implementation Techniques

The input of the Viatra2 model generation tool consists of one or more VTCL files, representing a Viatra2 model transformation. To examine the structure of this model transformation, the representing VTCL files need to be parsed. To this end, we used the Viatra2 Java API [48] provided by the Viatra2 model transformation framework [47]. The functionality of the Viatra2 model generation tool is implemented in Java.
Figure 7.5: Viatra2 metamodel (1)
Figure 7.6: Viatra2 metamodel (2)
Chapter 8

Validation

In this chapter we aim to answer research question RQ3, i.e., to what extent can the generic approach, as proposed in Chapter 5, replace the traditional approach of automated metrics extraction?

In the first section we discuss the approach that we applied for answering this question. Next, we present a semantically enriched Viatra2 metamodel. This language specific metamodel is used for the validation of the generic approach. In the subsequent section, we present the results of the validation. Finally, we summarize the insights that we acquired.

8.1 Validation Approach

The generic approach, as proposed in Chapter 5, can only be considered useful if it can (largely) replace the traditional approach of metrics extraction (see Section 5.1). Therefore, we need to determine to what extent the generic approach is appropriate for the extraction of a previously defined set of metrics. To this end, we decided to use the metrics defined in Chapter 4. The validation is considered successful in case more than 85% of the defined metrics can be extracted. In addition, the generated metric values must be correct.

In Section 7.1, we described the metrics extraction tool that implements the ideas presented in Chapter 6. Since we decided to validate the approach using the Viatra2 metrics defined in Chapter 4, the input of the tool must be a Viatra2 model and a semantically enriched version of its corresponding metamodel.

Appendix C includes a Viatra2 model transformation that serves as use case. It is specified in terms of a single VTCL file. Since the metrics extraction tool is developed to handle EMF (meta)models, this VTCL file needs to be transformed into an EMF model. For this purpose, we use the Viatra2 model generation tool described in Section 7.4.

Notice that the actual content of the model transformation is irrelevant for the validation of the generic approach. That is, the set of metrics generated by the metrics extraction tool is exclusively determined by the semantically enriched metamodel that is given as input. However, the generated metric values are determined by its corresponding model. To determine the correctness of these values, we compare them with manually extracted metric values.
8.2 Semantic Enrichment of Viatra2 Metamodel

This section describes how the Viatra2 metamodel can be semantically enriched to enable the interpretation of its corresponding model, i.e., with respect to the extraction of the metrics defined in Section 4. To this end, we use the semantic metaclasses described in Chapter 6. In the remainder of this section, we indicate their associations with the model elements of the Viatra2 metamodel. This language specific metamodel is depicted in Figures 7.5 and 7.6.

8.2.1 Construct Associations

A Viatra2 model transformation is specified in terms of one or more VTCL files. Furthermore, a VTCL file is specified in terms of standalone GT patterns, GT rules, standalone ASM rules and ASM functions. Various metrics are defined on these different levels of hierarchy. To extract these metrics, we need to explicitly indicate that these language constructs must be interpreted as constructs. Therefore, we associate language specific classes VTCLFile and Transformation with semantic class Construct.

Notice that class Transformation is a superclass of language specific classes GtPatternDef, GtRuleDef, AsmRuleDef and AsmFunctionDef. Alternatively, we could associate these subclasses with semantic class ContainerElement. However, in case all subclasses of a class must be equally interpreted, it is more concise to associate their superclass.

8.2.2 Id Associations

Standalone GT patterns, GT rules, standalone ASM rules and ASM functions are identified by their name and number of parameters (or arity). Since their name is not required to be unique, various metrics are defined that measure overloading. To extract these metrics, we need to explicitly indicate that their identifying names must be interpreted as Ids. Therefore, we associate language specific attribute name, which is part of class NamedElement, with semantic class Id.

8.2.3 Cyclomatic Complexity Associations

The Viatra2 model transformation language supports a number of iterative and branching ASM rules, viz., ASM iterate rule, ASM if rule and ASM try rule. Therefore, metrics are defined that measure the cyclomatic complexity. To extract the metrics, we need to explicitly indicate that these types of inline ASM rules must be interpreted as ‘cyclomatic complexities’. Therefore, we associate language specific classes IterateRule, IfRule and TryRule with semantic class CyclomaticComplexity. Furthermore, we give the concerning tags the identical name CyclomaticComplexity.

8.2.4 Dependency Reference Associations

Viatra2 model transformations are specified in terms of language constructs that can call each other, viz., standalone GT patterns, GT rules, standalone ASM rules and ASM functions. Various fan-in and fan-out metrics are defined that measure these dependencies. To extract these metrics, we need to explicitly indicate that the concerning references, which
are explicitly modeled in the Viatra2 metamodel, must be interpreted as dependency. Therefore, we associate semantic class \textit{DependencyReference} with the following language specific references:

- \textit{gtPatternDefCalls} (part of class \textit{PatternBody})
- \textit{asmRuleDefCall} (part of class \textit{RuleInvocationRule})
- \textit{gtPatternDefCall} (part of class \textit{ChooseRule})
- \textit{gtRuleDefCall} (part of class \textit{ChooseRule})
- \textit{gtPatternDefCall} (part of class \textit{ForallRule})
- \textit{gtRuleDefCall} (part of class \textit{ForallRule})
- \textit{gtPatternCall} (part of class \textit{Condition})

\textbf{8.2.5 (Dummy) Tree Reference Associations}

A GT pattern is, among others, expressed in terms of pattern bodies. A pattern body optionally contains an inline negative pattern, i.e., a \textit{neg}-keyword followed by a GT pattern. Recall that the use of inline negative patterns results in tree structures, i.e., GT patterns alternated with pattern bodies. Therefore, metrics are defined that measure these structures. To extract these metrics, we need to explicitly indicate that the concerning references must be interpreted as tree references. Therefore, we associate language specific references \textit{GtPattern-Def} and \textit{patternBodies}, which are part of classes \textit{Condition} and \textit{GtPatternDef} respectively, with semantic class \textit{TreeReference}. Since \textit{neg}-keyword should not contribute to the width and height of the formed tree structures, we associate language specific reference \textit{negativePatterns}, which is part of class \textit{PatternBody}, with semantic class \textit{DummyTreeReference}.

\textbf{8.3 Results}

This section presents the results of the validation. It describes to what extent the generic approach is appropriate for the extraction of the set of metrics defined in Section 4. The current implementation of the approach is unable to extract the complete set of metrics. In the remainder of this section, we elaborate on the subset of metrics that cannot be extracted.

\textbf{8.3.1 Combination of properties}

As indicated in Section 6.2.2, for each class in the containment hierarchy of an element, we measure for each of its (inherited) references the number of referred class instances. Three distinctions are made. In case a class contains an attribute, we distinguish among all possible attribute values. Furthermore, distinctions are made based on the references and subclasses of a class. Hence, distinct metrics are generated for all of its structural properties. However, no metrics are generated that measure the combination of multiple properties. Consequently, the following metric cannot be generated:
Considering combinations of properties could result in an immense number of metrics. Since we considered that this is undesirable, we restricted the metrics extraction tool to considering properties in isolation. Notice, however, that this restriction is implemented on purpose. Hence, it cannot be considered a restriction of the generic approach itself.

8.3.2 Unused parameters

Parameters can be unused for various reasons. To detect this, various metrics are defined. The current implementation of the generic approach is unable to detect unused parameters. To enable this detection, additional semantics need to be defined and implemented. The following enumeration gives an overview of the concerning metrics:

- # Unused parameters per GT pattern
- # Unused directed parameters per GT rule
- # Unused directed parameters per standalone ASM rule

8.3.3 Recursive GT patterns

GT patterns are either defined recursive or non-recursive. Similar to unused parameters, the current implementation of the generic approach is unable to distinguish between these types of GT patterns. To enable this distinction, additional semantics need to be defined and implemented. The following enumeration gives an overview of the concerning metrics:

- # Recursive GT patterns per VTCL file
- # Non-recursive GT patterns per VTCL file
- # Recursive inline GT patterns per GT rule
- # Non-recursive inline GT patterns per GT rule

8.4 Observations

The current implementation of the generic approach is unable to extract the complete set of metrics defined in Chapter 4. In total we defined 217 metrics. One of these metrics cannot be extracted due to a restriction of the metrics extraction tool, i.e., no metrics are generated that measure the combination of multiple properties. However, since this restriction is implemented on purpose, it cannot be considered a restriction of the generic approach itself. Furthermore, a number of metrics cannot be extracted due to the absence of required semantics, viz., unused parameters (3 metrics) and recursive GT patterns (4 metrics). Hence, 8 metrics cannot be extracted by the current implementation of the approach. However, about 96% of the defined metrics can be extracted.

To determine the correctness of the generated metric values, we compared them with manually extracted metric values. Since also the generated values appeared to be correct, the validation is considered successful.
Chapter 9

Conclusions and Future Work

This chapter presents the conclusions of the project. In addition, it suggests some directions for future work.

9.1 Conclusions

The conclusions of the project are presented by answering the research questions as stated in Section 1.2. The first research question that we aimed to answer was stated as follows:

RQ1. Can we apply direct quality assessment, as proposed in [38], to assess the internal quality of model transformations based on graph transformation?

As described in Chapter 4, we defined a set of metrics to assess the internal quality of model transformations specified in the Viatra2 model transformation language. Since metrics alone are not enough to assess quality, we related these metrics to quality attributes. Since the user community of Viatra2 was considered rather small, we identified these relations on the basis of our expectations. Although an empirical study is required to validate these relations, we are convinced that direct quality assessment can be applied to assess the internal quality of model transformations based on graph transformation. The answer to research question RQ1 is therefore stated as follows:

Answer RQ1. Yes, it is possible to apply direct quality assessment, as proposed in [38], to assess the internal quality of model transformations based on graph transformation. To increase the confidence of its suitability even further, empirical studies are required.

Since the development of metrics extraction tools is a time consuming task, and most of them have a considerable amount of overlap in functionality, we found it interesting to find out whether this task can be performed in a more generic fashion. The second research question that we aimed to answer was therefore stated as follows:

RQ2. Can we realize automated metrics extraction for model transformations using a generic approach?
In Chapter 5, we described two alternative approaches to the traditional approach of automated metrics extraction, i.e., developing a tool tailored towards the formalism at hand. The first approach that we considered was the use of an intermediate generic model. That is, an intermediate representation that abstract from language specific details. We assumed that abstracting from these details would reveal that model transformation formalisms provide similar language constructs. However, after studying a representative set of model transformation formalism, viz., Viatra2 [47], QVTO [18, 26], ATL [24] and Xtend [52], we observed that each of the formalisms also has some language specific concepts. Consequently, it was unclear what the intermediate representation should look like. Only preserving the language constructs the considered formalism have in common, would result in incomplete sets of metrics. Preserving all language constructs did not guarantee that the intermediate representation will be suited for other/future model transformation formalism as well. Therefore, we concluded that the use of an intermediate generic model is too restrictive.

The second approach that we considered was the semantic enrichment of a metamodel, representing a specific formalism, to enable the interpretation of its corresponding model. We observed that the advantage of this approach is that all functionality related to metrics extraction can be implemented language independently. The approach is therefore suited for the extraction of metrics defined for any model transformation formalism. In fact, the approach is not even restricted to model transformation formalisms, i.e., an arbitrary model can be interpreted, as long as its corresponding metamodel is semantically enriched. The answer to research question RQ2 is therefore stated as follows:

**Answer RQ2.** Yes, it is possible to realize automated metrics extraction (for model transformations) using a generic approach. The approach that we developed is suited for the automated extraction of metrics from an arbitrary model (transformation).

Since a generic approach can only be considered useful if it can (largely) replace the traditional approach of metrics extraction, the third research question that we aimed to answer was stated as follows:

**RQ3.** To what extent can the generic approach, as stated in RQ2, replace the traditional approach of automated metrics extraction?

In Chapter 8, we described to what extent the generic approach is appropriate for the extraction of the set of metrics defined in Section 4. We concluded that the current implementation of the approach is unable to extract the complete set of metrics. In total we defined 217 metrics. We observed that one of these metrics cannot be extracted due to a restriction of the metrics extraction tool, i.e., no metrics are generated that measure the combination of multiple properties. However, since this restriction was implemented on purpose, it cannot be considered a restriction of the generic approach itself. Furthermore, a number of metrics cannot be extracted due to the absence of required semantics, viz., unused parameters (3 metrics) and recursive GT patterns (4 metrics). We expect that these restrictions can be resolved by defining additional semantic metaclasses. Hence, 8 metrics cannot be extracted by the current implementation of the approach. However, about 96% of the defined metrics can be extracted. Furthermore, the generated metric values appeared to be correct. The answer to research question RQ3 is therefore stated as follows:
**Answer RQ3.** The current implementation of the generic approach, as proposed in Chapter 5, is unable to extract the complete set of metrics defined in Chapter 4. However, about 96% of the defined metrics can be extracted. We expect that most restriction can be resolved by defining additional semantic metaclasses.

### 9.2 Future Work

In section 4.4, we described the relations that we identified between quality attributes, as indicated in Section 4.2, and metrics, as described in section 4.3. These relations are exclusively based on our expectations. To validate these relations, and to assess whether the defined metrics are appropriate predictors for the quality attributes, an empirical study is required. This study would increase the confidence in the suitability of direct quality assessment, as proposed in [38], in the context of graph-based model transformations. In addition, the obtained observations could eventually be used to propose a methodology which, if adhered to, leads to high-quality model transformations [40].

In Section 8.3, we concluded that the current implementation of the generic approach, as proposed in Chapter 5, is unable to extract the complete set of metrics defined in Chapter 4. However, we expect that most of these restrictions can be resolved by defining additional semantic metaclasses. Furthermore, it would be desirable to enable the extraction of other frequently used measurements, e.g., Halstead complexity measures [21]. In addition, it is interesting to find out whether it is possible to enable the extraction of dependency metrics, without the need of explicitly modeled dependencies in the language specific metamodel.

Probably the most interesting question is whether the approach of semantically enriching a metamodel can also be applied for other purposed than metrics extraction. We expect that it is suited for other kinds of software analysis as well. Possibly, it can also be applied in a different problem domain.
Bibliography


<table>
<thead>
<tr>
<th>No.</th>
<th>Author(s)</th>
<th>Title</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>M.F. van Amstel, C.F.J. Lange, and M.G.J. van den Brand</td>
<td>Metrics for Analyzing the Quality of Model Transformations.</td>
<td><em>12th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering</em> (<em>QAOOSE 2008</em>), pages 41-51, Paphos, Cyprus, July 2008.</td>
</tr>
</tbody>
</table>


Appendix A

Metrics Extractor Guide

This appendix presents how the metrics extraction tool, described in Chapter 7, can be used for the automated extraction of metrics.

The input of the tool consists of an arbitrary model and its corresponding metamodel, representing a model transformation and its defining formalism, respectively. In addition, the semantic metamodel must be given as input. To this end, we implemented a simple user interface. Once the concerning files are selected, a metrics model can be generated by clicking on save metrics model. Figure A.1 depicts the user interface of the metrics extraction tool.

The source code of the metrics extraction tool can be found in the FALCON repository of the SET group, in a folder called MSc Project Roy.
Appendix B

Viatra2 Model Generator Guide

This appendix presents how the Viatra2 model generation tool, described in Section 7.2, can be used for the generation of Viatra2 models.

The input of the tool consists of one or more VTCL files, representing a Viatra2 model transformation. In addition, its corresponding VTML file must be given as input. Latter file contains the specification of the source (meta)models. Although these are not required for the generation of the Viatra2 models, the VTML file is required by the Viatra2 Java API [48].

The Viatra2 model generation tool can be executed in the Eclipse Environment, by running a file named MANIFEST.MF as OSGi framework. Its output consists of a Viatra2 model conforming to the Viatra2 metamodel (see Figures 7.5 and 7.6).

The source code of the Viatra2 model generation tool can be found in the FALCON repository of the SET group, in a folder called MSc Project Roy.
Appendix C

Viatra2 Model Transformation: Activity Diagrams to Petri Nets

This appendix includes a Viatra2 model transformation that describes a transformation from UML activity diagrams to Petri nets [48]. It is used in Chapter 8 for answering research question RQ3, i.e., to what extent can the generic approach, as proposed in Chapter 5, replace the traditional approach of automated metrics extraction?

```java
namespace ad2petri;

import uml.metamodel.uml;
import petrinet.metamodel;

machine Activity2PetriNet {
  // ENTRY POINT
  rule main() = call transform();

  rule transform() = seq {
  let Net = under in
    forall Activity with apply transformActivity(Activity, Net) do
      try seq {
        call checkTransformation(Activity);
        println("*** UML Activity " + name(Activity) +
          " has been transformed into Petri-net " + fqn(Net));
      }
    else println("*** Cannot not transform UML Activity: " + name(Activity));
    println("--- Transformation terminated.");
  }

  // TRANSFORM ACTIVITIES
}
```

Figure C.1: Activity Diagrams to Petri Nets (1)
grule transformActivity(out Activity, out PetriNet) = {
    precondition pattern unmappedActivity(Activity) = {
        'Activity' (Activity);
        neg find activityMapping(Activity, NoPetriNet);
    }
    postcondition find activityMapping(Activity, PetriNet)
    action {
        move(PetriNet, petrinet.models);
        call copyName(Activity, PetriNet);
        call activityTransformationSequence(Activity, PetriNet);
    }
}

pattern activityMapping(Activity, PetriNet) = {
    'Activity' (Activity);
    net(PetriNet);
    net.netTrace(Trace, PetriNet, Activity);
}

// TRANSFORMATION SEQUENCE OF ACTIVITY ELEMENTS

rule activityTransformationSequence(in Activity, in PetriNet) = seq {
    foreach X in Activity with apply transformControlFlow(X, PetriNet) do
        println("Transformed ControlFlow " + fqn(X));
    foreach X in Activity with apply transformInitialNode(X, PetriNet) do
        println("Transformed InitialNode " + fqn(X));
    foreach X in Activity with apply transformForkJoinFinalNode(X, PetriNet) do
        println("Transformed Fork/Join/FinalNode " + fqn(X));
    foreach X in Activity with apply transformExecutableNode(X, PetriNet) do
        println("Transformed ExecutableNode " + fqn(X));
}

// check whether anything remained that we could not transform
rule checkTransformation(in Activity) = {
    let Error = 0 in seq {
        foreach X in Activity with find untransformedActivityNode(X) do seq {
            println("ERROR - Could not transform activity node: " + fqn(X));
            update Error = 1;
        }
        foreach X in Activity with find untransformedActivityEdge(X) do seq {
            println("ERROR - Could not transform activity edge: " + fqn(X));
            update Error = 1;
        }
        if (Error > 0) fail:
    }
}

pattern untransformedActivityNode(ActivityNode) = {
    'ActivityNode' (ActivityNode);
    neg find activityNodeTransitionMapping(ActivityNode, PetriTransition);
}

Figure C.2: Activity Diagrams to Petri Nets (2)
pattern untransformedActivityEdge(ActivityEdge) = {
    'ActivityEdge'(ActivityEdge);
    neg find activityEdgeMapping(ActivityEdge, PetriPlace);
}

pattern activityNodeTransitionMapping(ActivityNode, PetriTransition) = {
    'ActivityNode'(ActivityNode);
    transition(PetriTransition);
    transition.transitionTrace(Trace, PetriTransition, ActivityNode);
}

pattern activityNodePlaceMapping(ActivityNode, PetriPlace) = {
    'ActivityNode'(ActivityNode);
    place(PetriPlace);
    place.placeTraceNode(Trace, PetriPlace, ActivityNode);
}

pattern activityEdgeMapping(ActivityEdge, PetriPlace) = {
    'ActivityEdge'(ActivityEdge);
    place(PetriPlace);
    place.placeTraceEdge(Trace, PetriPlace, ActivityNode);
}

pattern placeOfNet(PetriPlace, PetriNet) = {
    net(PetriNet);
    place(PetriPlace);
    net.places(Places, PetriNet, PetriPlace);
}

pattern transitionOfNet(PetriTransition, PetriNet) = {
    net(PetriNet);
    transition(PetriTransition);
    net.transitions(Transitions, PetriNet, PetriTransition);
}

pattern transitionPlaceArc(PetriTransition, PetriPlace) = {
    transition(PetriTransition);
    transition.inArc(InArc, PetriTransition, PetriPlace);
    place(PetriPlace);
}

pattern placeTransitionArc(PetriPlace, PetriTransition) = {
    transition(PetriTransition);
    place.outArc(OutArc, PetriPlace, PetriTransition);
    place(PetriPlace);
}

// MISC. TRANSFORMATION HELPERS

Figure C.3: Activity Diagrams to Petri Nets (3)
pattern placeMarked(PetriPlace, Token) = {
  token(Token);
  place(PetriPlace);
  place.marking(Marking, PetriPlace, Token);
}

pattern placeOfOutgoingEdge(ActivityNode, PetriPlace) = {
  'ActivityNode'(ActivityNode);
  'ActivityNode'.outgoing(OutGoing, ActivityNode, ActivityEdge);
  'ActivityEdge'(ActivityEdge);
  place.placeTraceEdge(Trace, PetriPlace, ActivityEdge);
  place(PetriPlace);
}

pattern placeOfIncomingEdge(ActivityNode, PetriPlace) = {
  'ActivityNode'(ActivityNode);
  'ActivityNode'.incoming(InComing, ActivityNode, ActivityEdge);
  'ActivityEdge'(ActivityEdge);
  place.placeTraceEdge(Trace, PetriPlace, ActivityEdge);
  place(PetriPlace);
}

grule connectNodeToOutgoing(in ActivityNode, in PetriTransition, out EdgePlace) = {
  precondition find placeOfOutgoingEdge(ActivityNode, EdgePlace)
  postcondition find transitionPlaceArc(PetriTransition, EdgePlace)
}

grule connectNodeToIncoming(in ActivityNode, in PetriTransition, out EdgePlace) = {
  precondition find placeOfIncomingEdge(ActivityNode, EdgePlace)
  postcondition find transitionPlaceArc(EdgePlace, PetriTransition)
}

rule copyName(in SrcUML, in TrgPetri) =
  let NameStr = undef, NameEdge = undef in seq {
    rename(TrgPetri, name(SrcUML));
    new (datatypes.'String'(NameStr) in TrgPetri);
    new (namedElement.name(NameEdge, TrgPetri, NameStr));
    setValue(NameStr, name(SrcUML));
  }

// TRANSFORM Control Flow

grule transformControlFlow(out ControlFlow, in PetriNet) = {
  precondition pattern unmappedControlFlow(ControlFlow) = {
    'ControlFlow'(ControlFlow);
    neg find activityEdgeMapping(ControlFlow, NoPetriPlace);
  }
  postcondition pattern mappedControlFlow(ControlFlow, PetriPlace, PetriNet) = {
    'ControlFlow'(ControlFlow);
    find activityEdgeMapping(ControlFlow, PetriPlace);
  }
}

Figure C.4: Activity Diagrams to Petri Nets (4)
Figure C.5: Activity Diagrams to Petri Nets (5)
find placeMarked(PetriPlace, Token);
find placeTransitionArc(PetriPlace, PetriTransition);
find activityNodeTransitionMapping(ActivityNode, PetriTransition);
find transitionOfNet(PetriTransition, PetriNet);
}

action {
call copyName(ActivityNode, PetriTransition);
rename(PetriPlace, name(PetriTransition)+"Activations");
forall EdgePlace with apply
    connectNodeToOutgoing(ActivityNode, PetriTransition, EdgePlace) do skip;
}
}

// TRANSFORM Fork/Join/Finish Nodes

grule transformForkJoinFinalNode(out ActivityNode, in PetriNet) = {
precondition pattern unmappedForkJoinFinalNode(ActivityNode) = {
ad2petri.helpermetamodel.forkJoinFinalNode(ActivityNode);
noq find activityNodeTransitionMapping(ActivityNode, NoPetriTransition);
}
postcondition pattern mappedForkJoinFinalNode(ActivityNode, PetriTransition, PetriNet) = {
ad2petri.helpermetamodel.forkJoinFinalNode(ActivityNode);
find activityNodeTransitionMapping(ActivityNode, PetriTransition);
find transitionOfNet(PetriTransition, PetriNet);
}
}

action {
call copyName(ActivityNode, PetriTransition);
forall EdgePlace with apply
    connectNodeToOutgoing(ActivityNode, PetriTransition, EdgePlace) do skip;
forall EdgePlace with apply
    connectNodeToIncoming(ActivityNode, PetriTransition, EdgePlace) do skip;
}
}

Figure C.6: Activity Diagrams to Petri Nets (6)