SEMI-AUTOMATED TEST MODEL GENERATION

Saheed Olatunji Popoola
MSc (by Research)

University of York
Computer Science
March 2016

Abstract

Appropriate test models that can satisfy complex constraints are required for testing model management programs in order to build confidence in their correctness. Models have inherently complex structures and are often required to satisfy non-trivial constraints which makes them time consuming, labour intensive and error prone to construct manually. Automated capabilities are therefore required, however, existing fully-automated model generation tools cannot generate models that satisfy arbitrarily complex constraints. This thesis addresses this problem by proposing a semi-automated approach towards the generation of such models. A new framework named Epsilon Model Generator (EMG) that implements this approach is presented. The framework supports the development of model generators that can produce random and reproducible test models that satisfy complex constraints.

Contents

Acknowledgements

Firstly, I am most grateful to my supervisor, Dr. Dimitris Kolovos for his invaluable support and feedback throughout the duration of this degree. Without his competent advice and supervision, I could never have completed this work. I am also grateful to my assessor, Dr. Fiona Polack for her support and advice. I'm indebted to Horacio Rodriguez and other colleagues in the Enterprise Systems group for their fruitful contributions.

My warmest gratitude goes to my parents, Yemisi Popoola (nee Gbolahan) and Bolanle Popoola, my partner, Barakat Ojewale and all my friends for the mental and emotional support they have offered me throughout these months.

I'm also grateful for the financial support of National Information Technology Development Agency (NITDA), Nigeria under its scholarship development programme.

Finally, all praise is due to Allah, the lord of the worlds, the most beneficent , the most merciful.

Author Declaration

I declare that all parts of this thesis, except where explicit reference is made to the contribution of others, is the result of my own research. Parts of this work have been submitted to an international conference. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

1 Introduction

Model Driven Engineering (MDE) makes use of models as first class artefacts for software development. The structure of these models varies depending on their intended usage and properties. The structure of models is described by metamodels and sometimes further external constraints might be imposed on the models so that they can exhibit other desired characteristics not captured by the metamodels. In an MDE process, models are manipulated by automated model management programs in the context of activities such as model-to-model transformation, model validation, model composition, etc. Such programs can have defects (bugs) and they therefore need to be tested. In order to test them we need appropriate test data, i.e models that conform to the respective metamodel and satisfy the required constraints. The application of MDE to larger and more complex systems makes the structure of the metamodels complex and harder constraints are needed in order to satisfy the intended purpose. In this context, manual assembly of such test models is error prone, time and labour consuming, hence there is a need to automate the generation process.

1.1 Motivation

This section summarises the current state of the art in automated model generation and highlights the problem that motivated the work in this thesis.

1.1.1 Models in Model Driven Engineering

A model [1] can be defined as an abstraction of a phenomena of interest. In Model Driven Engineering (MDE), models are defined using a set of well-defined rules and semantics which are encapsulated in what is called their metamodels [2]. Therefore, a metamodel is a model that defines another model. The metamodel usually enforces some constraints in order to specify the model it defines. However, due to the complexity of the phenomena described by a model, its metamodel may not be able to fully capture all the properties the model is expected to have, therefore additional external constraints may also be necessary. A valid model is required to exhibit the characteristics specified by its metamodel and satisfy any additional external constraints.

Traditionally, models have been used in software development mainly for documentation and they were rarely updated when changes were made to the code. MDE makes use of models as first class artefacts to be used throughout the software development process. This makes it easier to focus on the software design without considering

the underlying computing environment or programming language, thereby reducing software development time.

Two types of modelling languages are generally recognised: general purpose and domain-specific languages [3]. General purpose languages such as UML [4] generally capture a wide range of problems and provide constructs that can be used for diverse applications. Domain-specific languages are designed to be used in a narrow range of problems and therefore provide constructs that are restricted to a particular domain.

1.1.2 Model Management Programs

MDE inherently relies on mechanisms that can be used to execute different operations on models. These operations are termed model management operations. A model management program is a software program that can be used to manipulate models automatically (e.g. validate, compare, transform, merge). As MDE is increasingly applied to larger and more complex systems, the models that the model management programs need to manage also grow significantly in complexity and are required to satisfy increasingly more complex constraints. Within the context of this thesis, complex constraints include constraints that involves string literals, multiple first-order OCL operations [5] or compound associations between the metamodel elements' types [6]. A simple constraint does not have any of the afore-mentioned features.

1.1.3 Model Generation

Appropriate test data is essential for testing software programs in order to build confidence on their correctness. The test data required for testing model management programs is models. Models inherently have complex structures, yet this complexity is further amplified because they are often required to satisfy non-trivial constraints. Manual generation of such models is error prone, labour intensive and time consuming therefore automated generation of test models is needed. An ideal model generator should be able to:

1. Generate models that conform to a target metamodel and respect additional (complex) constraints.

2. Exhibit other desired characteristics such as randomness and repeatability of the models produced.

Several state-of-the-art model generation tools have been evaluated using an existing complex model transformation as a case study and none of them were able to generate models that satisfy the constraints which are pre-conditions of the transformation. This situation has motivated the research presented in this thesis.

1.2 Research Contributions

A new approach that simplifies the development of model generators has been proposed, and a framework named Epsilon Model Generator that implements the proposed approach has been developed. The framework is a semantic extension to an existing language named Epsilon Pattern Language (EPL) [7] and provides first-class support for common tasks in a model generation process. The framework has been used to develop model generators that can generate repeatable and random models that satisfy complex constraints.

1.3 Thesis Structure

Chapter 2 provides a detailed review of related work. Section 2.1 introduces the concept of MDE and presents various types of model management programs of interest. The section also reviews languages and tools for specifying and executing such programs. Section 2.2 discusses various approaches to testing software programs and the different types of testing that currently exist. Section 2.3 discusses major barriers towards testing of automated model management programs and the significance of an appropriate model generation process is highlighted. Section 2.4 examines important requirements to be fulfilled by an ideal model generator and reviews existing automated generation approaches, their merits and limitations.

Chapter 3 provides a detailed analysis of current model generation tools based on the approaches identified earlier and highlights their weaknesses particularly in generating models that satisfy complex constraints as well as their inability to reproduce the generated models. Chapter 4 summarizes the findings of the literature review and the limitations of existing approaches and tools. The chapter also states the objectives of this research and how we intend to achieve them. Chapter 5 introduces a new approach towards generating models and presents a model generation framework based on the approach and built on top of the Epsilon platform. Section 5.4 gives a practical demonstration of how the framework can be used to generate models that are repeatable, random and easy to parameterize.

Chapter 6 evaluates the proposed framework and shows how it meets the research objectives stated in Chapter 4. The final Chapter summarises the work done and outlines possible future research directions.

2 Literature Review

This chapter provides a comprehensive review of related work in the field of automated model generation within the context of Model Driven Engineering (MDE). The aim of Section 2.1 is to introduce the core MDE concepts to the reader with a focus on technologies that will be used in the remainder of the project (e.g. Eclipse Modeling Framework, Epsilon). An extensive review of all the different dimensions of MDE (e.g. different metamodelling frameworks , model-to-model and model-to-text transformation languages etc.) is beyond the scope of this section as this work only contributes to a specific facet of MDE (automated test model generation). Section 2.2 discusses various approaches to testing software programs. Section 2.3 discusses the process of testing model management programs and also highlights the significance of using appropriate test models. Section 2.4 examines current approaches towards automated model generation, their merits and limitations.

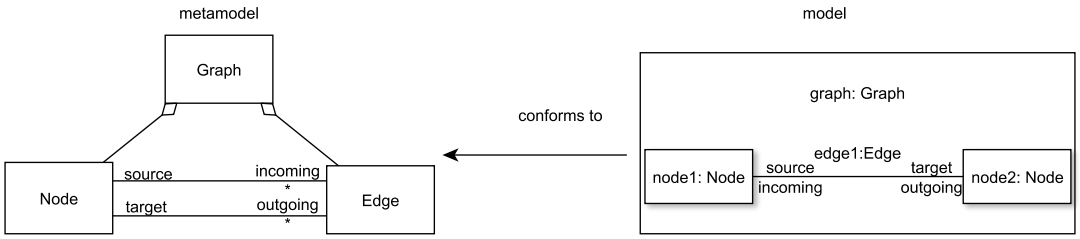
2.1 Model Driven Engineering

Models [1] have been used since the inception of the field of software engineering with notations such as flowcharts and finite state machines due to their usefulness in simplifying the complexity of software systems and communicating technical aspect of the software development process to non-technical audiences [2, 8, 1]. In a traditional software engineering process, models are mainly used for documentation and communication of software design. Occasionally, they may also be used for partial code generation but the developer has to modify and complete the code using traditional programming languages. The models were seldom updated as changes applied to the code are rarely reflected back in the models.

MDE elevates models as first class artefacts to be used throughout the software development lifecycle. This enables developers to focus on the problem space rather than on the underlying computing environment [9]. Problems can therefore be expressed in terms of concepts in the problem domain such as health, automotive, etc. By providing support for capabilities such as automated code generation, MDE reduces the gap between models and code thereby eliminating the need for their manual translation which can be error-prone and time consuming.

The concepts in a problem space are abstracted in *models* which are described using what is called their *metamodels* [2]. Metamodels are models that describe the concepts in each domain and the relationships among them. A conformance relationship exists

Figure 2.1: A Metamodel and its Conforming Model



between a model and a metamodel hence, a model is said to conform to the metamodel when each of its elements' type is in the metamodel and the relationship among the model elements follows the defined relationships among the corresponding types in the metamodel. A conforming model is also a valid instance of the metamodel.

Figure 2.1 presents an example of a directed graph metamodel (in MOF [10]) and a model that conforms to it. The metamodel describes that a directed graph is composed of nodes which are connected by edges. Each edge connects exactly two nodes: one as a source and the other as a target. The source node treats the edge as an incoming connection while the target node treats it as an outgoing connection. Sometimes it may be necessary to add further external constraints to the models so that they can exhibit additional characteristics. For example we may add the following constraint to the graph metamodel: there should be two special nodes, one that connects via only outgoing Edges and the other connects via only incoming Edges.

2.1.1 Modelling Technologies

This thesis deals with generation of models that can be used for testing model management programs. This section introduces modelling technologies that are useful for specifying models while Section 2.1.2 discusses model management programs and existing frameworks for executing them.

Modelling technologies are important for specifying and constructing models. Some of the state-of-the-art modelling technologies include: Microsoft Software Factories [11] which provides a set of APIs for constructing models in Microsoft Visual Studio and NetBeans Metadata Repository [12], a framework for constructing, storing and managing MOF 1.4 compliant models. MetaEdit+ [13] is a commercial modelling technology that provides a collaborative environment for managing models. As discussed above, a comprehensive survey of different modelling technologies is beyond the scope of this section, hence below we focus on the modelling technologies used in this work.

Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [14] is possibly the most commonly used modeling framework. EMF is a modeling extension to the Eclipse IDE[1] and provides capabilities for creating, deleting and querying models and model elements via a set of APIs. At the core of EMF is an object-oriented metamodeling language called Ecore which can be used to describe metamodels and also provides capabilities for runtime change notification and persistence support via XML serialization. The Ecore metamodel is its own metamodel i.e it is a meta-metamodel.

Graphical Modeling Framework

The Graphical Modeling Framework (GMF) [15] is an Eclipse-based framework that provides runtime capabilities for generating graphical editors using EMF and the Graphical Editing Framework (GEF) [2]. GEF provides the necessary infrastructure for creating rich graphical editors and views therefore, GMF can be seen as a bridge that links the EMF to GEF at runtime. GMF requires as input an Ecore metamodel, a graphical notation model, a tooling model and a mapping model that links the first three models.

2.1.2 Model Management Frameworks

Model management frameworks and tools are useful for specifying and executing model management programs i.e software programs that can manipulate models by automatically executing model management operations. Common operations that can be performed on models are stated below.

1. Model to model transformation: This is the process whereby a model is transformed into another model corresponding to the same or to a different metamodel.

2. Model to text transformation: This is the process of transforming a model into textual artefacts. It forms the basis of code generation and attempts to close the gap between models and code similar to the role of compilers in translating high level code to executable instructions.

3. Model validation: This is the process of verifying if a model conforms to its metamodel and satisfies a set of constraints.

4. Model merging: This is the process of combining two or more models into a single model.

5. Model comparison: This is the process of comparing models and identifying common elements. It consists of two main tasks: *model matching* which identifies common elements and *model differencing* which highlights the differences in the models using the results of the previous task.

[1] eclipse.org
[2] http://www.eclipse.org/gef/

6. Model migration: This deals with automated updating of a model due to changes in its metamodel.

Presently, there are several model management frameworks and tools. They include: ATL Transformation Language (ATL) [16], Acceleo [17], Xtend [18], VIATRA [19], etc. but they will not be reviewed in detail here because, for the purpose of this work, model management programs will be treated as black boxes. However, we will briefly examine the Epsilon framework [7] because it is used in the remainder of this work.

Epsilon Framework

The Epsilon framework [7] is a family of languages for different types of model management programs. The framework provides a layer of abstraction that enables diverse modelling technologies (such as EMF) to be manipulated in a uniform manner. The following languages currently exist in the Epsilon framework:

1. Epsilon Object Language (EOL) [20]: This is the core of the Epsilon framework. It is a computationally complete language that provides a set of reusable model management facilities on top of which task specific languages can be implemented. It can also be used as a standalone language for generic model management.

2. Epsilon Pattern Language (EPL) [7]: EPL matches patterns in a model based on the rules specified using the language. Pattern matching is an important step in many model management operations hence, EPL was designed to be easily integrable with diverse modelling technologies and other model management programs.

3. Epsilon Transformation Language (ETL) [21]: The language was designed for hybrid model to model transformation. It takes as input an arbitrary number of source models and transforms them into an arbitrary number of target models corresponding to the same or different metamodels. ETL also supports diverse modelling languages and technologies.

4. Epsilon Validation Language (EVL) [22]: This provides model validation support to the Epsilon framework. It can be used to specify and verify complex constraints on models.

5. Epsilon Wizard Language (EWL) [23]: This is used for automating repetitive tasks during model management operations.

6. Epsilon Generation Language (EGL)[24]: EGL provides model to text transformation capabilities to the Epsilon platform.

7. Epsilon Comparism Language (ECL) [25]: ECL is useful for comparing model elements between two models of potentially different metamodels and/or modelling technologies.

8. Epsilon Merging Language (EML) [26]: EML contributes capabilities for carrying out model merging operations.

9. Epsilon Flock for Model Migration [27]: This provides support for efficient model migration operations.

Each language was designed to effectively manage a particular model management task. Some languages such as EPL and EWL are not bound to a specific model management task but provide generic support that can be used in executing other tasks. All the languages of the Epsilon platform are built on top of the EOL which provides a uniform set of reusable operations. This helps to remove unnecessary complexities such as diverse syntax and duplication of code.

2.2 Software Testing

Having introduced the core concepts of MDE in Section 2.1, this section provides a brief overview of software testing approaches and types, in preparation for Section 2.3 which presents an overview of existing work on testing model management programs and highlights the importance of generating appropriate models - which is the topic of this thesis.

Software testing is the process of analysing a software system in order to detect bugs or evaluate its features [28, 29]. A bug is an error in the system that makes the system produce unexpected results or behave in an unwanted manner. The presence of bugs in a system has led to considerable costs in form of correction and sometimes the development of a new system [30]. For example in 2012, Knight Capital Group[3] lost about 440 million dollars in 30 minutes due to bugs in its trading software[4]. Therefore it is important that computer system should be adequately tested in order to boost the level of confidence in their correctness. Different types of bugs [31, 32] to be detected include:

1. Inability of a program to produce the expected output.
2. Failure of a program for input that satisfies its preconditions (in this case either the program needs to be fixed or its preconditions need to be strengthened).
3. An incorrect step in the program.
4. A human mistake.
5. Inability of the program to interoperate with other software and hardware.

2.2.1 Approaches to Software Testing

There are three main approaches to software testing. In one approach which is known as black-box testing [33, 34, 35], the tester does not have any special knowledge about

[3]https://www.kcg.com/
[4]http://fortune.com/2012/08/02/why-knight-lost-440-million-in-45-minutes/

the internal structure and implementation of the system that is being tested. The tester only knows what the system is expected to do and not how it does it. They use their intuition and experience to select suitable test cases that are likely to detect bugs in the system. This approach works best for verifying that the system performs according to its specification e.g detection of incorrect functionality, performance errors, etc. [36, 37].

Another approach is white-box testing [38, 34, 35], where the tester has detailed knowledge of the inner workings of the system. The tester uses this knowledge, to produce test cases that can detect bugs e.g test cases that will go through error prone code paths, or code paths that are only executed in exceptional circumstances. This approach works best for verifying that the system performs its operation in the correct way e.g detection of invalid data structures, etc. [36, 35].

A third approach known as grey-box testing [34, 39] is a combination of black-box and white-box approaches. The tester has only a partial knowledge of the inner workings of the system that makes it easy for them to design suitable test cases while still ignorant of the main execution process.

2.2.2 Types of Software Testing

There are different types of software testing that are designed for different purposes. They include:

1. Destructive testing [40]: This is a type of software testing whereby the system is tested with different varieties of test cases with an intent of making the system fail. It can be used to verify the performance of the system when it receives unexpected or incorrect input thereby establishing its robustness. This form of testing is used, for example, by the Android Monkey tool [5].

2. Stress testing [41]: In this type of testing, the system is subjected to an intense pressure usually above its normal working capacity and to its break point in order to determine its maximum workload.

3. Regression testing [42, 36]: This is used to verify that a modified system that has been previously tested still performs correctly after the modification. The main objective is to verify that new changes to the system did not create new bugs.

4. Performance testing [43, 44]: This type of testing verifies how a system performs under a particular workload. It is used to ascertain some properties of the system such as its scalability or reliability.

5. Fault injection [45, 46, 47]: This is a form of testing whereby a bug is intentionally added to the system to verify if the bug will be detected or not. It is used to ascertain the robustness of the testing procedure.

[5] http://developer.android.com/tools/help/monkey.html

6. Mutation testing [48, 49]: This type of testing aims to verify the robustness of the testing process. A slightly modified variant of the system called mutant is automatically generated using some (mutation) operators. The testing process is then executed to assess if the changes can be detected.

2.3 Testing of Model Management Programs

Automated model management programs are software programs, therefore they can have bugs. The testing of these programs is therefore important in order to build confidence in their correctness [50]. The various approaches to testing a traditional software discussed in Section 2.2 is also applicable when testing model management programs. However, model management programs and the required input data (test models) are inherently complex which makes the testing process more difficult. [51] outlines three main barriers to testing of model management programs. They are test data generation, adequacy criteria and oracle construction.

Test Data Generation

Constructing test data that satisfy a set of criteria is an important requirement in any software testing operation [52, 53, 54]. The test data required in the testing process of model management programs are models [55]. Models usually have complex structures which makes them error prone and time consuming to create manually therefore, automated capabilities are required.

Adequacy Criteria

A typical model management program typically has an infinite number of input models that it can process. Therefore, only a representative sample of the expected input models can be used during a testing process. A major challenge is how to determine if a specific set of models is enough to test all the requirements the program is expected to fulfil and to detect any bugs, if present [56]. A common approach is to employ the service of a seasoned tester who is familiar with common bugs and the models that are appropriate to test for their presence. Another approach is the application of additional constraints to the test data in order to boost the level of confidence in the testing process [51, 57] e.g Every class in the models' metamodel must be instantiated at least once in each of the models.

Oracle Function

This is a function to determine if the result of the testing operation is correct [58]. If an expected model is available, the generated model can be compared with the expected model [25, 59]. For example, to test model transformation or merging programs, the oracle is an expected model while, for model validation it is a set of constraints the models are expected to pass or fail. However, if an expected model is not available,

a partial oracle that satisfies some specified properties may be constructed [60]. For example , in the generation of random models for a model to model transformation, the expected outcome of the random generation process is unknown, therefore the test oracles would be to determine if the transformation process would produce a runtime exception in the transformation or satisfaction of constraints. An exception would indicate a failure while a pass is indicated by the absence of any exception. This is a form of destructive software testing because it is aimed at causing the system to fail.

2.4 Model Generation

As shown in Section 2.2 and 2.3, the generation of test models is central to the testing of model management programs when using a black-box testing approach. In white-box testing, the tester has access to inner workings of the system and the necessary models are usually manually crafted for the intended purpose. But in black-box testing approach, the tester only knows what the system does and not how it does it. Hence in order to test the functionalities, different kinds of models are required therefore there is need for model generation. Once an oracle function has been constructed and the test adequacy criteria have been defined, the generation of test models that meet these requirements is essential. Manual constructions of these models is time consuming and error prone hence, there is a need to automate the process of generating these models. [55, 61] highlights some important criteria that should be fulfilled by an automated model generator. They are:

1. Validity: Generate models that conform to their metamodels and respect any constraints imposed on them.

2. Adequacy: The number of models or model elements generated should meet the adequacy criteria.

3. Scalability: Able to generate models with many model elements.

4. Flexibility: The generator should be easy to parameterize so that different kinds of models can be generated for different testing scenarios.

We would like to add a new criterion called *repeatability* and this is the ability of the model generator to reproduce exactly the same test models generated previously. This is important if there is a need to reproduce a bug or for independent confirmation. It is also useful in ensuring that developers don't have to exchange - potentially large - models to reproduce problems across different machines, in which case the generation process may simply be repeated instead of transferring the models.

Several approaches are found in the literature on how to automate model generation and they can be grouped into four main classes.

Constraint Satisfaction: This is the most common approach found in the literature and has been used in [62, 63, 64, 65]. In this approach, the metamodel and constraints are transformed into a Constraint Satisfaction Problem (CSP) or a Satisfiable Modulo Theory (SMT). These problems are then solved using a CSP or SMT solver and the resulting solutions which represent valid instances of the metamodel are transformed back to a model format. A major challenge confronting this approach is automatic constraint solving since the constraint problems are usually heterogeneous and complex. This approach is flexible and produces valid models but it can only handle simple constraints [66, 67, 68, 55].

Model Fragmentation: In this approach, which has been implemented by [52], a partial model known as *model fragment* is manually created and the remaining part of the model is automatically generated based on the information in the model fragment. The major motivation is that most of the times, there are specific properties the user is interested in when generating a model, however these properties have some other dependencies that must be satisfied in order to produce a valid model. This approach enables the user to focus on the objects of interest i.e the model fragments, and the rest are then automatically generated. This approach is flexible, but not suitable for large models with many elements. It cannot handle external constraints and substantial human intervention is required due to the need for model fragments [52, 69, 61].

Configuration: The Configuration approach [70, 71] transforms the metamodel and the constraints into a configuration model e.g grammar, with some rules which are determined by the relationship of the types in the metamodel. Model elements are first generated and the rules are then used to determine the kind of relationship that should exist among them. External constraints are not considered during the generation phase but a constraint checker (e.g Prover9 [72]) may be used to filter the valid models. This approach is scalable but it may produce invalid models [69, 66, 55].

Tree: In the Tree approach [61], the metamodel is represented as a tree specification by mapping the classes and relationships in the metamodel to nodes and edges in the specification. Large random trees corresponding to the tree specification are then generated using the Boltzman algorithm [73]. These trees are then transformed back to a model format. This approach is suitable for generating large numbers of models but the generated models may be invalid [66, 69].

In summary, the Constraint Satisfaction approach is not scalable and can only handle simple constraints while the Model Fragment approach requires substantial human effort as the interesting parts of the test model are constructed manually. Both the Configuration approach and the Tree approach do not consider external constraints, therefore they may be unable to generate valid models. None of these approaches considers the repeatability of the generated test models. Table 2.1 provides a summary

Table 2.1: Summary of Approaches to Automated Test Model Generation

Approaches	**Valid**	**Adequate**	**Scalable**	**Flexible**
Constraint satisfaction	Yes	?	No	Yes
Model fragmentation	?	Yes	No	Yes
Configuration	No	Yes	Yes	No
Tree	No	No	Yes	No

of our review of existing approaches. A question mark (?) denotes that such criterion cannot be verified from the literature for the specific approach.

2.5 Chapter Summary

This chapter has motivated the need for automated model generation as well as the criteria an ideal model generator is expected to fulfil. The different approaches that are currently being used to automatically generate models were discussed and for each approach, its merit and limitations were highlighted. This review has identified two main research problems: existing approaches to model generation do not support complex external constraints. Also repeatability of the generated models, an essential requirement for reproducing a testing process was not considered in the state-of-the-art approaches.

3 Assessment of Existing Tools

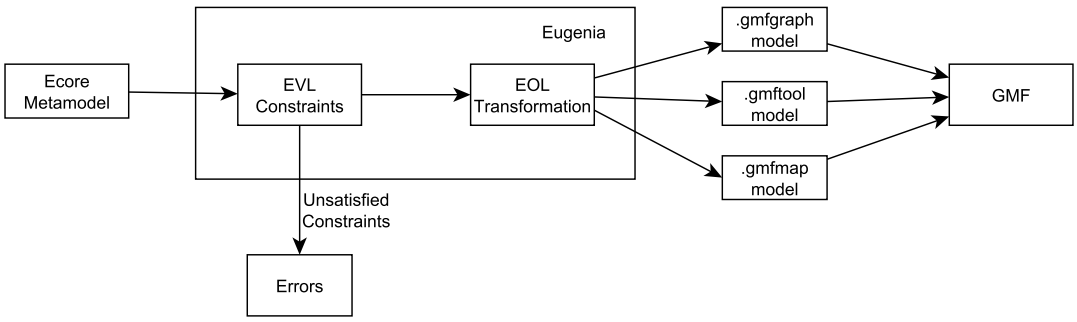
This chapter provides a comprehensive assessment of existing fully-automated model generation tools. Section 3.1 presents our running example, a real-world complex model transformation that requires input metamodels that satisfy a set of non-trivial constraints. Section 3.2 examines existing tools based on the approaches identified earlier and establishes their inability to generate models that can satisfy sets of non-trivial constraints.

3.1 Running Example: Eugenia

Eugenia [74] was chosen as a running example because it is a complex transformation that pre-dates this research and requires its input metamodels to satisfy complex constraints. Eugenia is a tool that transforms an appropriately annotated Ecore metamodel into a set of models from which the Eclipse Graphical Modelling Framework [15] can generate a complete graphical editor for instances of the metamodel. The input Ecore metamodel must satisfy a set of Eugenia-specific constraints (e.g. the "@gmf.diagram" annotation needs to appear in exactly one class in the metamodel) before the transformation can be executed. Figure 3.1 provides an overview of the Eugenia transformation process.

A typical Eugenia transformation process involves two model management operations; a model validation and a model transformation. The model validation process checks if the input Ecore metamodel satisfies the Eugenia-Specific constraints while the model transformation process adds a graphical editor for models that conforms

Figure 3.1: Eugenia Framework



to the metamodel. In total, Eugenia requires its input Ecore metamodels to satisfy 26 constraints (364 lines of code) specified using Epsilon Validation Language (EVL) [22]. Listing 3.1 illustrates three of these constraints.

Listing 3.1: Subset of Eugenia Constraints

```
context EPackage {
  constraint DiagramIsDefined {
   check: getDiagramClass().isDefined()
   message: 'One class must be specified as gmf.diagram'
  }
  constraint ContainmentReferencesAreDefined {
   guard : self.satisfies('DiagramIsDefined')
   check : getDiagramClass().getContainmentReferences().size
       () >0
   message : 'Diagram class ' + getDiagramClass().name + '
       must define ' + ' at least one containment reference'
  }
  constraint NodesAreDefined {
   guard: self.satisfies('DiagramIsDefined')
   check: getNodes().size()>0
   message: 'No nodes (gmf.node) have been defined'
  }
}
```

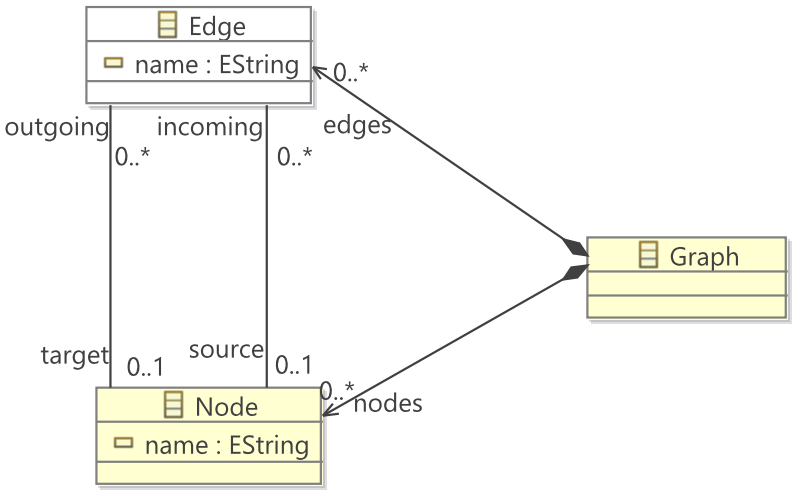
Constraint "DiagramIsDefined" specifies that exactly one EClass should be annotated as "gmf.diagram". Constraints "ContainmentReferencesAreDefined" and " NodesAre-Defined" state that the EClass annotated as "gmf.diagram" must also have at least one containment reference and a reference to an EClass that has been annotated as "gmf.node" respectively. Appropriate error messages are produced if any of the constraints are not satisfied. Listing 3.2 is an example of an appropriately annotated Ecore metamodel expressed in the Emfatic textual notation [1].

Listing 3.2: Eugenia-annotated Ecore model in Emfatic

```
@namespace(uri="graph", prefix="graph")
package Graph;
@gmf.diagram
class Graph {
  val Node[*] nodes;
  val Edge[*] edges;
}
@gmf.node(label="name")
class Node {
  attr String name;
  ref Edge[*]#source incoming;
  ref Edge[*]#target outgoing;
}
```

[1] http://www.eclipse.org/modeling/emft/emfatic/

Figure 3.2: A Sample GMF Editor




```
14  @gmf.link(label="name", source="source", target="target",
        style="dot", width="2")
15  class Edge {
16    attr String name;
17    ref Node#incoming source;
18    ref Node#outgoing target;
19  }
```

The complete set of constraints along with a short explanation of each of the constraints is available in Appendix A while an OCL translation is provided in Appendix B. Existing model transformation tools do not support EVL constraints therefore an OCL translation is needed to assess existing tools. A direct translation from EVL to OCL was carried out during the course of this work for each of the Eugenia constraint. The translation is straightforward and its faithfulness has been evaluated through testing.

If an input Ecore metamodel satisfies these constraints, the model transformation process is expected to generate the required set of models (gmfgraph,gmftool and gmfmap) for GMF to produce a graphical editor for instances of the input metamodel. The generated models are produced based on the annotations that are specified by the Eugenia constraints. For example, the class annotated as ”gmf.diagram” is assumed to be the root object while classes annotated as ”gmf.node” are shown as nodes in the generated diagram. However since exactly one class must be the root object, constraint ”DiagramIsDefined” is used to enforce this. Similar constraints are used to describe required properties needed for an efficient transformation. Figure 3.2 is a sample GMF editor for the Ecore metamodel in Listing 3.2.

In order to test Eugenia, there is a need to automatically generate Eugenia-annotated Ecore models (as test cases) and try to identify cases where models satisfy the tool’s

additional 26 constraints but cause the transformation to fail. These tests are intended to either reveal missing constraints or bugs in the transformation.

3.2 Assessment of Existing Fully-Automated Model Generation Tools

This section reports our findings on the ability of state-of-the-art fully-automated tools to generate models that can satisfy the Eugenia constraints discussed in Section 3.1. Nine Ecore-based tools were identified in the literature, however only five of them were available at the time of writing this thesis. The performance of these tools in generating models that satisfy complex constraints was examined and their ability to reproduce exactly the same generated models was also assessed. The available tools are:

1. Grimm [55]: This is a tool developed based on the Constraints Satisfaction approach. It takes as input an Ecore based metamodel and an optional set of OCL constraints which are translated into instances of Constraint Satisfaction Problems (CSP) and solved using the Abscon solver [75].

2. EMFtoCSP [64]: This tool implements the Constraints Satisfaction approach to model generation. The tool translates Ecore based metamodels and OCL constraints into CSP instances which are solved using the ECL^{i}PS^e solver[2]. Other properties such as partial or full satisfaction of the constraints and constraint redundancies e.g. if a constraint depends on another constraint, can also be verified.

3. Cartier (Pranama) [65] and Alloy [76] : This tool was previously called "Cartier" but has now been renamed as "Pranama". It is based on a Constraints Satisfaction approach and automatically translates an Ecore based metamodel into a Kermata-based [77] metamodel. The Kermata metamodel is then translated into instances of Satisfiable Modulo Theory (SMT) problems along with additional OCL constraints which are then solved using an Alloy solver. The OCL constraints currently require manual translation into Alloy SMT problems which are then added to the ones translated from the Kermata metamodel. The solutions are then automatically transformed into the required model format.

4. RMG [71]: This tool implements a Configuration approach towards model generation. It takes as input an Ecore based metamodel and a set of constraints which are specified using a graphical interface provided by the tool. The metamodel and the constraints are then transformed into a configuration model which is used to guide the generation process. However, since the constraints have to be entered using a graphical interface, only specific types of constraints supported by the interface can be imposed on the model. The constraints supported include number of instances of model elements and a range of primitive values for attributes

[2] http://eclipseclp.org/

while assignment of specific values to attributes and existential quantifiers are not supported.

5. MM2GRAGRA[70]: This tool uses a Configuration approach in which an XMI based metamodel is translated into instances of graph grammar using a graph transformation tool called Attributed Graph Grammar System (AGG)[3]. A restricted form of OCL constraints can be translated as graph constraints which are added to the automatically generated instances of the graph grammar . The instances of the graph grammar and the graph constraints are used by the AGG tool to guide the generation process of an arbitrary number of models.

The unavailable tools include:

1. ASMIG [78] : A Small Metamodel Instance Generator (ASMIG) is a tool that implements the Constraints Satisfaction approach towards model generation by translating an Ecore based metamodel and a set of OCL constraints into SMT problems using an attributed graph notation [79]. The SMT problems are then solved by SMT solvers that support the SMT-Lib V2 specification[4] such as Z3 SMT solver [80], Mathsat5 [81] or SMTInterpol [82]. The tool is presently available[5] but documentation on how to use it is not yet ready.

2. Trust [62]: This tool uses a combination of Constraint Satisfaction approach and search techniques to generate models. The metamodel and a set of OCL constraints is translated into a CSP which is solved by a solver that uses search techniques to generate valid instances.

3. Omogen [52]: This tool implements a Model Fragmentation approach towards model generation. It takes as input a metamodel and a set of model fragments. Instances of the metamodel are then generated based on the elements in the model fragments. The developers of the tool noted that it cannot handle external constraints.

4. Tree Spec [61]: The tool was developed based on the Tree approach with a focus on the scalability of the models generated i.e models with millions of model elements. A metamodel is transformed into a tree specification and then huge number of instances are generated using Boltzmann algorithm [73]. However, the evaluation of additional external constraints to the metamodel was not fully considered.

3.2.1 Assessment Process

The first task was to assess whether the available tools were able to generate Ecore models (instances of Ecore.ecore metamodel) without any additional constraints. None of the tools were able to generate instances of the metamodel because they do not

[3]http://user.cs.tu-berlin.de/ gragra/agg
[4]http://smtlib.cs.uiowa.edu/
[5]https://bitbucket.org/classciwuhao/asmig/overview

support all the features in the Ecore metamodel. Although not all features of the Ecore metamodel are required in order to generate models that satisfy the Eugenia constraints, the tools failed because they attempted to instantiate every class and feature present in the metamodel even when it was not necessary.

A simplified version of the Ecore metamodel was then developed and all the tools were able to generate models conforming to this metamodel without any additional constraints. However, none of the tools was able to reproduce the exact models generated because they do not provide support for reproducing generated models. The Eugenia constraints were then translated into different formats supported by each tool (specified in Table 3.1) and added as input to the simplified Ecore metamodel. None of the tools were able to produce a valid model that satisfies the Eugenia constraints.

Grimm produces a "constraints is unsatisfiable" error because it does not support the "exists" feature of OCL and assignment of specific values to attributes therefore constraints such as "DiagramIsDefined" which specifies that exactly one class should be annotated as "gmf.diagram" could not be satisfied. EMFtoCSP stopped responding and the program was terminated after about 2 hours. The program was re-executed many times and the tool becomes unresponsive during each execution cycle. The cause of this behaviour could not be determined and execution process had to be manually terminated after few hours. RMG's (graphical) constraint language is not expressive enough to specify the Eugenia constraints because it lacks support for bounded constraints, existential quantifiers etc. thereby making it impossible to specify most of the constraints. Pranama, formerly called "Cartier", was combined with an Alloy CSP solver but Alloy does not currently support string literals. This makes it impossible for the tool to satisfy constraints such as "DiagramIsDefined" and "NodesAreDefined" which requires some classes to be annotated with a specified string literal i.e "gmf.diagram" and "gmf.node" respectively. MM2GRAGRA also produces an error because it does not support string literals. Table 3.1 summarises the findings of this exercise.

Table 3.1: Analysis of Automated Model Generation Tools

Tool	**Input Meta-model**	**Con-straints**	**Approach**	**Output**	**Reason for Failure**
Grimm	Ecore	OCL	Constraints	Error: constraint is unsatisfiable	Does not support OCL function ”exists”
EMFto CSP	Ecore	OCL	Constraints	Non-deterministic	Hangs
RMG	Ecore	Graph-ical	Configura-tion	Constraints cannot be translated to RMG specification	Graphical constraint language not expressive enough
Pranama & Alloy	Ecore	OCL	Constraints	Error	String literals not supported
MM2GR AGRA	AAG	AAG	Configura-tion	Error	String literals not supported

3.3 Chapter Summary

This chapter has examined existing model generation tools and their ability to generate models that can satisfy complex constraints. This chapter has validated the research problems identified in Chapter 2 by demonstrating that: none of the tools based on existing approaches were able to generate models that can satisfy complex constraints such as Eugenia constraints. Also repeatability of the generated models, an essential requirement for reproducing a testing process was lacking in these tools.

4 Analysis and Hypothesis

Through the review of related work in Chapter 2 and the assessment of existing tools in Chapter 3, a number of challenges were identified. This chapter further analyses these challenges and then establishes the research hypothesis and objectives of this thesis. Finally, the research methodology we intend to follow in order to meet these objectives is outlined.

4.1 Research Challenges

This section summarizes the research challenges identified in Chapters 2 and 3.

Automated model management programs usually manipulate models that can satisfy complex constraints, therefore it is essential that for testing such programs, models that satisfy such constraints are used as input test data. Existing fully-automated tools have been shown to be unable to produce complex models that would be necessary to test real-world model transformations (such as Eugenia).

For scenarios in which fully-automated model generators fail, the alternative is for developers to either construct test models manually or write bespoke model generators using a programming/model management language such as Java, QVTo [1] or EOL [83]. Manual construction of such models is error-prone, labour intensive and time consuming. Developing a bespoke model generator from scratch is a challenging endeavour as developers need to think about important properties from first principles. The properties are stated below.

1. Expressiveness of the language selected to develop model generators is important in order to be able to construct any arbitrarily complex model. Most programming/model management languages are usually computationally complete, hence they are suitable for developing model generators that can generate complex models.

2. Readability makes a generator easy to understand and enhances user-friendliness [84].

3. Conciseness, which is the ability to develop a model generator with minimal syntactic constructs, is necessary to improve maintainability [85].

[1] https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

4. Performance is important for producing models with large number of model elements in reasonable time. The increasing complexity of modern software has steadily led to models with large number of model elements, hence performance is often an important consideration.

5. Randomness is necessary in a testing process in order to minimize bias.

6. Reproducibility is necessary for remote/independent confirmation of results.

An ideal model generation process should be expressive, readable, concise, high-performing, random and reproducible.

4.2 Research Hypothesis

With regards to the challenges presented above, the context of the research hypothesis is as follows:

As MDE is increasingly applied to larger and more complex systems, models that are manipulated by model management programs also increase in complexity and the constraints that they are required to satisfy become harder. Testing such programs requires models that satisfy complex constraints as input test data. Manual assembly of such models using a general purpose language is error prone, labour intensive and existing fully-automated model generation approaches fail in generating models which satisfy complex constraints.

In this context, the hypothesis of this thesis is stated below:

A dedicated language for model generation can be used to develop more concise, configurably random and reproducible model generators that perform better than generators constructed with a general purpose language without sacrificing expressiveness and readability.

The objectives of this research are to:

1. Identify recurring patterns during the model generation process.

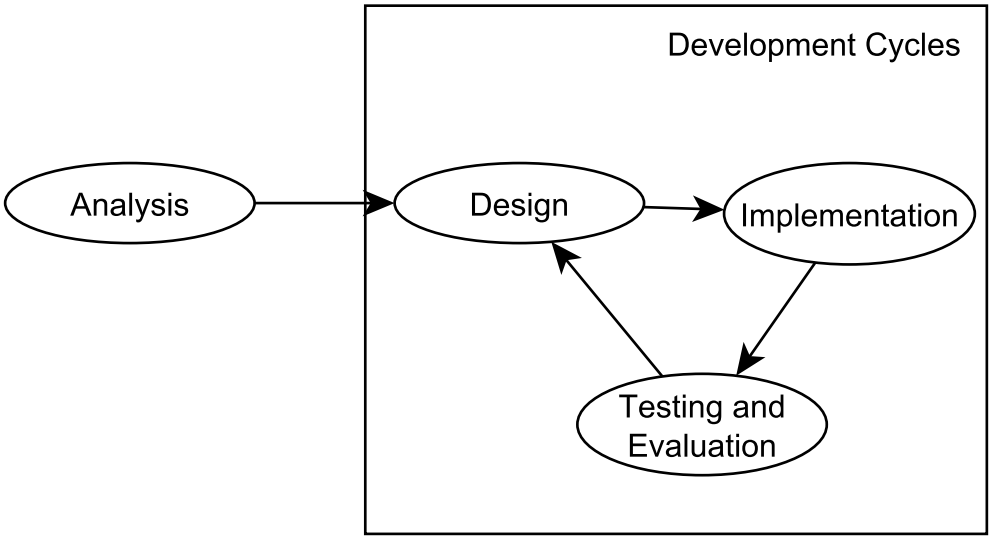
2. Encode the identified patterns in the syntax of a dedicated language.

3. Evaluate the validity of the hypothesis beginning with conciseness and ability to reproduce models.

4.3 Research Scope

A decision has been made to limit the scope of this work to the generation of EMF-based models. EMF is a typical 3-layer metamodelling architecture and as such it is expected that the results of this research are trivially portable to any other similar architecture.

Figure 4.1: Overview of research methodology



4.4 Research Methodology

An iterative process has been followed in order to evaluate the hypothesis. This includes an initial analysis phase followed by design, implementation, testing and evaluation cycles. Figure 4.1 provides a graphical overview of the process.

4.4.1 Analysis

A review of existing work on automated model generation has been conducted in the analysis phase. The features, strengths and shortcomings of available tools have been evaluated using a non-trivial case-study. A number of research challenges that have motivated the research hypothesis and objectives of this thesis were also identified.

4.4.2 Design and Implementation

Based on the findings of the analysis phase, a new approach towards model generation has been conceived to investigate the hypothesis. This approach has been implemented in the form of a framework that simplifies the development of model generators that can produce random and repeatable models.

4.4.3 Testing and Evaluation

Several model generators have been developed using the framework to assess how well they exhibit desired properties. A comparison with similar model generators

constructed using a general purpose language has also been carried out in order to validate the research hypothesis.

4.5 Chapter Summary

This chapter summarized the challenges identified in the literature review. The hypothesis and objectives of this thesis were also stated and the methodology we intend to follow in order to meet these objectives was also discussed.

5 Semi-Automated Model Generation

This chapter discusses recurring tasks in developing a bespoke model generator and then introduces a semi-automated approach to model generation, which is aimed at model generation scenarios for which fully-automated solutions currently fail. The Epsilon Model Generation (EMG) framework that implements this approach is also presented.

5.1 Recurring Patterns in a Bespoke Model Generation

A bespoke model generator is usually developed using a programming/model management language such as Java, QVTo[1] or EOL [83] for model generation scenarios that cannot be handled by fully-automated tools. However, developing a bespoke model generator from scratch is a challenging endeavour as developers need to think about properties such as reproducibility, randomness and flexibility from first principles. Randomness is necessary to reduce bias while reproducibility is essential for repeating a generation scenario which may be required due to a fault in the process or for confirmation of results. Flexibility, which in this context refers to the ability to configure the size of generated models, is important so that the generator can be adapted for diverse purposes (e.g. correctness/peformance testing). In general, model generation involves three recurring tasks.

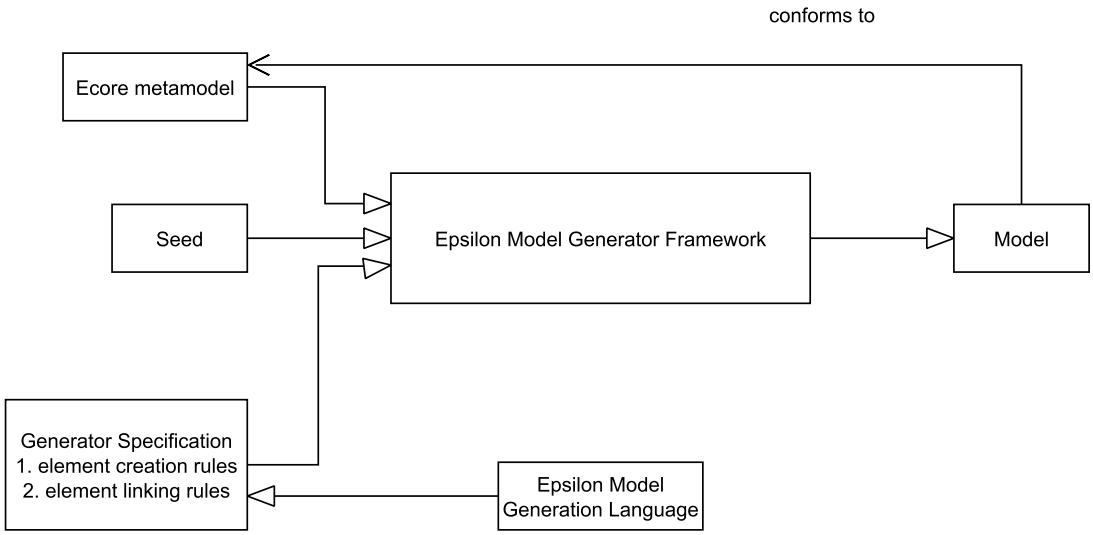
1. Creation of model elements. For example, in producing a Graph model that conforms to the Ecore metamodel such as the one produced in Section 5.6.1, model elements of type Graph, Node and Edge need to be created. Two common subtasks associated with this task are also identified: specifying the number of instances of each element type that should be created and (optionally) identifiers for the elements created.

2. Generation of appropriate (random) values and assignment of these values to the attributes of the model elements.

3. Linking the model elements together so that the generated model conforms to the metamodel and satisfies any additional constraints.

[1] https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

Figure 5.1: Overview of the Generation Framework



5.2 Epsilon Model Generation Framework

The Epsilon Model Generation (EMG) framework has been developed within the context of this work to simplify the development of model generators. The framework is built on top of the Epsilon platform [83] and implements a semi-automated approach to model generation by automating recurring tasks in a model generation process. EMG leverages an existing Epsilon language (Epsilon Pattern Language [7]) to support the development of model generators that fulfil the following requirements:

Randomness Generate random models that conform to an Ecore-based metamodel.

Parameterization Characteristics of these models (e.g how many instances/type, values for features) are easily parametrized.

Repeatability Generated models are reproducible.

Figure 5.1 is a graphical overview of the generation framework. The framework takes as input an Ecore-based metamodel, an optional "seed" parameter and model generation rules written in the *Epsilon Model Generation language (EMG)*, a language developed within the context of this work and explained in detail in Section 5.4. The expected inputs are:

1. Ecore-based metamodel: This is the metamodel of the models to be generated. The types of model elements in the generated model must be present in the metamodel.
2. Seed: This is an optional (integer) parameter that is used to ensure repeatability of the models generated. If the same seed is supplied, it is expected that the framework would be able to reproduce a model that has been generated previously.

However, if no seed is supplied, the framework randomly produces a seed which may be used in the next generation cycle.

3. Generator specification: This is a set of rules that governs the generation process and contains two types of rules: *element creation rules* for generating model elements and *element linking rules* which determine how the model elements should be linked together in order to conform to the specifications in the metamodel and satisfy any required constraints.

Since this is a semi-automated approach, it should be stressed that the responsibility for ensuring that generated models conform to the Ecore metamodel and satisfy the required constraints lies with the developer of the model generation rules.

5.3 Epsilon

The EMG framework is built on top of the Epsilon platform which has been briefly introduced in Section 2.1.2. In particular, the EMG language extends the Epsilon Pattern Language (EPL) [7] and makes use of Epsilon Object Language (EOL) operations [20] These languages (EPL and EOL) are discussed in detail because most of their features were reused in EMG.

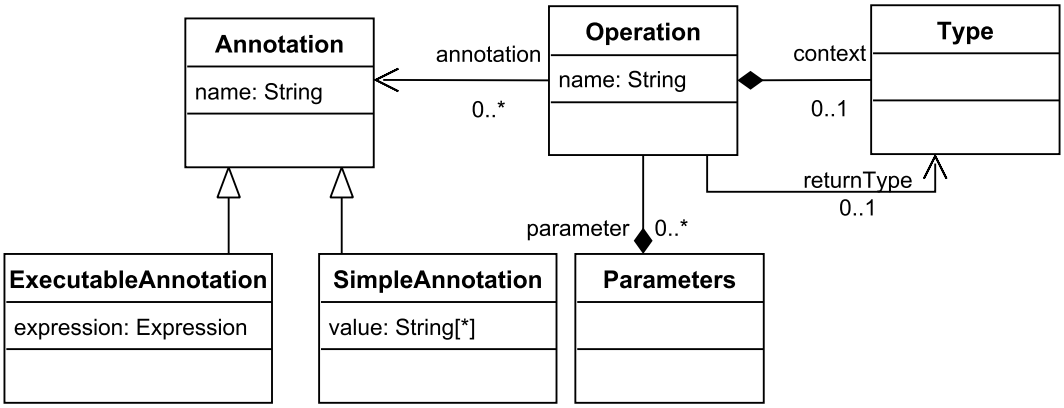
5.3.1 Epsilon Object Language

The Epsilon Object Language (EOL) [20] is the core of the Epsilon framework and provides a set of reusable components that simplify the development of other task-specific languages. EOL programs are organised in modules that contain a body and a number of user-defined operations. An EOL operation typically has a type, a name, a return type and a set of annotations as shown in Figure 5.2. The type specifies the context (instances of a type in the metamodel or an in-built type) of the operation's execution, the name is the operation's identifier and the return type specifies the context of the return value. Annotations may also be added to provide additional information about the operation. Two types of annotations are supported: *simple annotations*, whose values are determined at compilation time and *dynamic annotation* whose values are computed at runtime. A simple annotation is denoted with the prefix @ while a dynamic annotation is denoted with the prefix *$*. EOL also supports polymorphism whereby multiple operations may have the same name and parameters with unique context types.

Listing 5.1: A sample EOL operation

```
1 @cache
2 operation Graph isEmpty(): Boolean{
3   if(self.nodes.isEmpty()){
4     return self.edges.isEmpty();
5   }
6   return false;
7 }
```

Figure 5.2: Abstract Syntax of an EOL Operation [7]

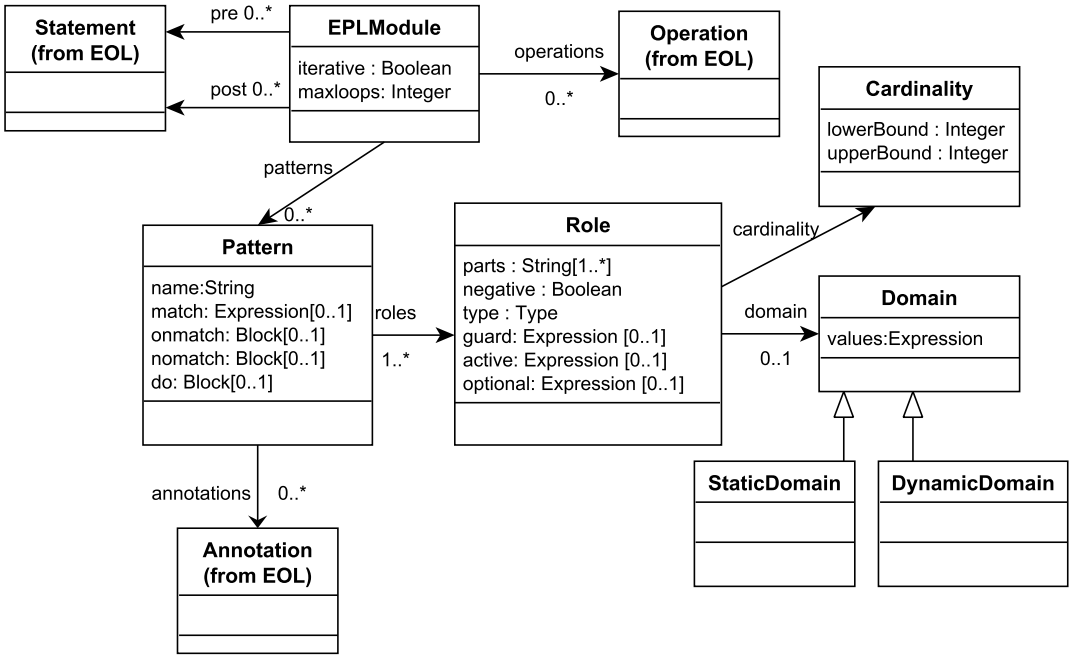


Listing 5.1 is a sample EOL operation named ”isEmpty” that checks if a Graph is empty (i.e it has no node or edge) and returns object of type ”Boolean”. The context type for this operation is ”Graph” meaning that this operation can be called on objects of type ”Graph”. The operation has no parameter and checks if the nodes of the Graph are empty using an inbuilt operation called isEmpty which checks if the size of a collection is 0. If the nodes are empty, the operation then checks if its edges are empty. This operation has a simple annotation named ”cached” to indicate that it is a cached operation i.e the operation is only executed once: subsequent calls return the same result without executing its body again.

5.3.2 Epsilon Pattern Language

Epsilon Pattern Language (EPL) [7] is a language for specifying and identifying instances of patterns among model elements. The language is organised into modules and it is computationally complete because it extends EOL which is a computationally-complete language [86, 87]. An EPL module contains a number of patterns, EOL operations and optional pre and post EOL statement blocks that are executed before and after the execution of the main process respectively. Figure 5.3 gives an overview of the abstract syntax of an EPL module [7].

Figure 5.3: Abstract Syntax of an EPL Module [7]

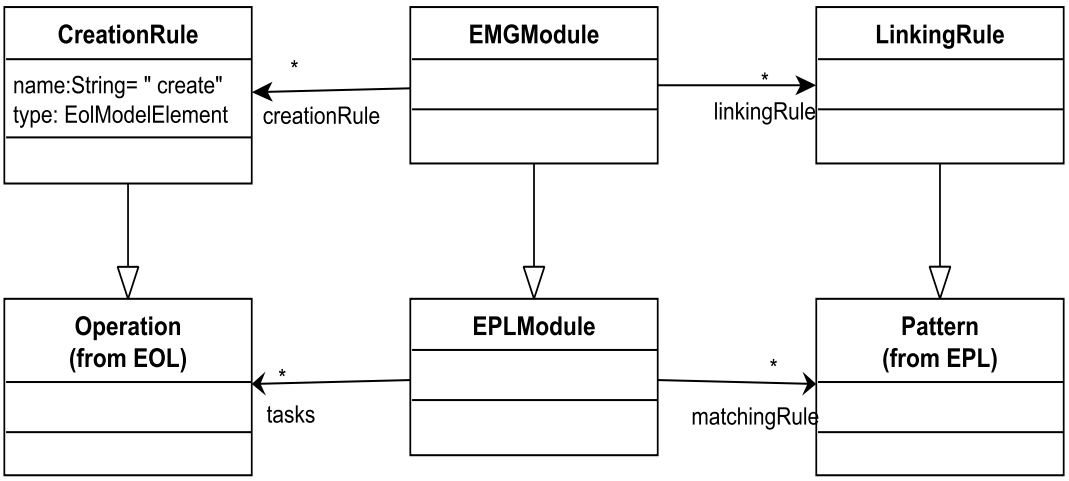


Listing 5.2: A sample EPL pattern

```
1  pattern linkNode
2    source:Node,
3    target:Node,
4    edge:Edge
5    from: (Edge.all.select(E|E.name.isDefined()))
6    guard: edge.source.isUndefined() and edge.target.isUndefined(){
7      onmatch{
8        edge.source=source;
9        edge.target=target;
10     }
11   }
```

Listing 5.2 is a sample EPL pattern named ”linkNode” that links two objects of type ”Node” together using an object of type ”Edge”. The two Nodes are named ”source” and ”target” while the Edge is named ”edge”. A domain is specified for the ”edge” using the keyword ”from” which indicates that only Edges whose names are defined are considered for the matching process. This domain is further restricted using the ”guard” notation by specifying a condition to be satisfied i.e the Edge should be unused for connecting either source or target nodes. No domain or guard is specified for the Nodes; all available Nodes are therefore considered for the matching process. The ”onmatch” notation specifies the action to be executed if a match is found and in this

Figure 5.4: Abstract Syntax of EMG



sample i.e it connects two matching Nodes together by linking them to an Edge.

5.4 Epsilon Model Generation (EMG) language

The Epsilon Model Generation (EMG) language was developed within the context of this work to specify model generation rules. The language is a semantic extension to EPL whose syntax was not changed but its execution semantics has been altered to better fit the problem of model generation. New built-in operations and annotations have been added and support for desirable properties such as repeatability has been provided. Changes to the execution semantics of EPL include: altering all random functions to have a common seed (for repeatability) and ensuring that special EOL operations (used for creating model elements) are always executed before the linking (this means all model elements are usually created before starting the linking process).

The frequent activities in a model generation process identified in Section 5.1 have been abstracted into language constructs in EMG. As such, an EMG program is composed of two types of rules, *creation rules* for producing model elements and *linking rules* for connecting them. Creation rules produce a configurable number of model elements and an optional identifier associated with them. Linking rules provide support for specifying groups of elements to be linked together and how they should be linked. Annotations are used to add more information on how the rules should be executed e.g the ”instances” annotation associated with a creation rule is used for specifying the number of elements to be produced. In-built operations provide support for generating and assigning values to model elements while additional user-defined tasks can be automated using standard EOL operations. Figure 5.4 provides a graphical overview

of the abstract syntax of the EMG language. A generator specification is organized as an EMGModule, an extension of EPLModule that contains EOL operations and EPL patterns. In an EMGModule, creation rules are EOL operations named ”create” and have a context of type EOLModelElement (i.e instances of a type in the metamodel) while the linking rules are EPL patterns.

5.4.1 Creation Rules

These are used to create model elements and assign values to the elements' attributes. A creation rule is specified by an EOL operation named ”create” whose context type indicates the type of model elements to be created. Two EOL annotations may be added to this operation. They are:

1. $instances: This is a dynamic annotation named ”instances” and its expected value is an integer or a range of two integers to indicate the number of model elements to be generated. If an integer is assigned, the exact number of elements is produced but if its value is a range, then a random number of elements between the boundaries of the range (inclusive) is produced. If no value is assigned or the annotation is not specified, only one instance is created. A dynamic annotation is used because if the specified value is a range, the actual value (a random number between the range's boundaries) is computed at runtime.

2. @name: This is a (optional) simple annotation named ”name” and its expected value is a String which is used as an identifier for grouping the model elements created by the operation. If multiple operations specify the same value, all the model elements created from such operations are grouped together and identified by the value of the annotation.

5.4.2 Linking Rules

Our initial consideration was to use EOL operations to implement linking rules. However, we found out that most of the support we would need to add to EOL operations in order to optimize them for linking model elements already exists in EPL patterns. Hence a decision was made to specify the linking rules by EPL patterns with additional in-built operations and annotations for linking created elements. An EPL Pattern comprises several roles, each of which has a name, a type, a domain (from) and a guard. The guard specifies additional constraints to be satisfied by candidate elements. An EPL pattern also has an ”onmatch” attribute that specifies the actions to be executed when matches are found. Additional information can also be provided on a pattern by using annotations.

An additional built-in operation is provided for linking elements created using creation rules. The operation named ”getCreatedElements(value:String)” has a String argument and returns a collection of objects that were generated by the creation rules associated with the identifier specified by the ”value” of the operation's argument.

Specific annotations may also be added to the patterns and they are:

1. @probability: This is a simple annotation named ”probability” and it specifies the probability that a successful match will be returned. A probability of 1 indicates that the ”onmatch” actions are executed for all successful matches while a probability of 0 would not execute these actions for any successful match. This is important in cases where elements can only be linked together if they satisfy some constraints but the satisfaction of the constraints does not require that the elements should be linked.

2. @noRepeat: This is a simple annotation named ”noRepeat” and it specifies that once a match has been found, the matching elements would no longer be considered within the same context for a possible match by other unmatched elements.

3. @number: This is a simple annotation named ”number” that indicates the maximum number of matches required. When this number of matches is found, the matching process of that pattern is stopped.

5.4.3 Code Reusability

New tasks may be implemented using standard EOL operations so that the implementation code can be reused many times in the same/different EMG program. However these operations should not be named ”create” so as to be differentiated from a creation rule.

5.4.4 Randomness

Several in-built random operations that can be used to ensure randomness of the structure of the model and to generate random values to be assigned to model elements have been added to EMG. These operations can generate objects of primitive types or select a random object from a collection. They are listed in Table 5.1.

5.4.5 Repeatability

The same random value generation facility is used throughout the generation process to ensure that the generated model can be reproduced. The random generator produces values for attributes of the model elements and also guides the matching process using the indexes of the newly-created model elements. The seed of this random value generator may be specified using the runtime configuration tool provided with this framework (shown in Figure D.3) when a particular generated model needs to be reproduced or it may be randomly generated by EMG for new models.

5.4.6 Parameterization

In order to improve the flexibility of the model generators, some aspects e.g number of instances of model elements to be generated can be represented as parameters whose

Table 5.1: Additional In-built Operations

Signature	**Context Type**	**Description**
uniRandomD(): Any	Any	Selects a random object from a collection of objects using a uniform distribution
uniRandom (size:Integer): Collection	Collection	Returns a collection with ″size″ objects from a collection of objects using a uniform distribution
binRandom():Any	Collection	Selects a random object from a collection of objects using a binomial distribution
binRandom(mean:Integer, variance:Integer) :Any	Collection	Selects a random object from a collection of objects using a binomial distribution with a mean and variance
expRandom(): Any	Collection	Selects a random object from a collection of objects using an exponential distribution
randomString(): String	Any	Generates a random string of a random length between 5 and 20
randomString(exp:String):String	Any	Generates a random string based on the regular expression specified in the argument
randomString (low:Integer, high:Integer):Any	Any	Generates a random string of a random length between the value ″low″ and ″high″
randomInteger(): integer	Any	Generates a random integer
randomInteger (low:Integer, high:Integer): Integer	Any	Generates a random integer between the numbers ″low″ and ″high″
randomReal(): Real	Any	Generates a random real number
randomReal (low:Real, high:Real):Real	Any	Generates a random real number between ″low″ and ″high″
random-Boolean():Boolean	Any	Generates a random boolean.
randomBoolean (num:Real): Boolean	Any	Generates a random boolean with the probability of generating a true value being ″num″ where num is between 0 and 1 (inclusive)

values are provided at runtime. This makes it easy to generate models of different

structures and sizes without changing the specification rules. For example the number of nodes to be generated may be represented by a parameter called ”nodeCount”. This is implemented in EMG by assigning ”nodeCount” as a value to the ”$instances” annotation. A value of 5 may then be assigned to the parameter ”nodeCount” at runtime to indicate that 5 Nodes should be created.

5.4.7 Completeness

The EMG language extends EPL which is a computationally complete language. Hence, EMG can be used to express arbitrarily complex model generation logic.

5.5 Generation of Models

EMG simplifies the development of bespoke model generators by using a 2-step approach to generate models based on recurring patterns in a typical model generation process that was identified in Section 5.1. The 2-step approach separates the generation of model elements and linkage of the generated elements. In the first step, model elements are generated using the creation rules while the second step links the model elements based on the linking rules. This 2-step approach ensures that all the elements are equally considered during the linking phase as opposed to creating and linking them simultaneously. Secondly, it also ensures that arbitrarily complex constraints, which may require linking up yet-to-be-created model elements (if both steps were done simultaneously) can be expressed in EMG.

EMG adopts a semi-automated approach, hence the generation rules need to be written manually using the EMG language. Therefore, the correctness of the generated models lies with the developer of the model generation rules.

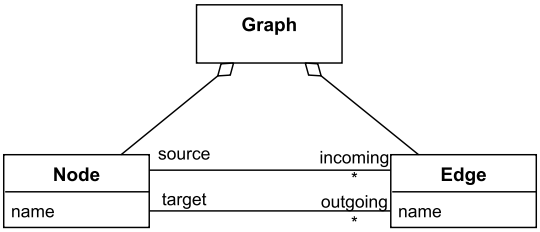
5.6 Sample model Generators

This section presents two sample generators that have been developed using EMG as running examples to demonstrate the interplay and usefulness of the features discussed above. The first generator generates Graph models with simple constraints with the second generator generate models that satisfy Eugenia constraints.

5.6.1 Graph

Consider the case in which we want to generate models that conform to the Graph metamodel, where each Graph contains N *maxNodes* which are connected by Edges, where N is a random Integer greater than two. Each Node is also expected to be connected to exactly two other Nodes; one as incoming and the other as an outgoing connection. Figure 5.5 is the metamodel of the models to be generated and it is the same as the one shown in Figure 2.1.

Figure 5.5: Graph Metamodel



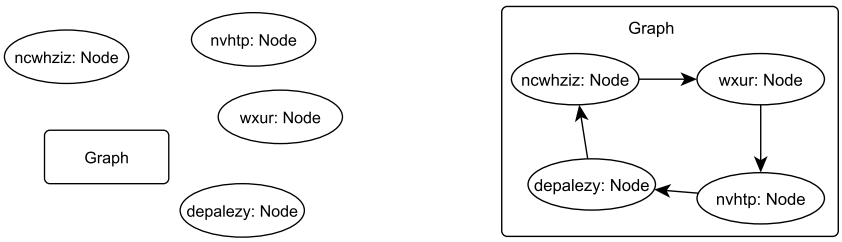
Listing 5.3 displays an EMG program that contains rules for generating models that conform to the Graph metamodel and satisfy the additional constraints.

Listing 5.3: EMG Program to Generate a Graph Model

```
1  operation Graph create(){
2  }
3  $instances Sequence{2,maxNodes}
4  operation Node create(){
5    self.name= randomString();
6  }
7  pattern linkNodes
8  graph:Graph,node:Node
9    guard: node.incoming.isEmpty(){
10     onmatch{
11       var edge: new Edge;
12       edge.source= node;
13       edge.target=
14       Node.all.select(n|n.outgoing.isEmpty).uniRandom();
15       graph.nodes.add(node);
16       graph.edges.add(edge);
17     }
18   }
```

Two creation rules (lines 1 to 6) and one linking rule (lines 7 to 18) have been specified. The create operation for Graph (line 1) creates a single element of the type ”Graph”. Lines 3 to 6 create the Nodes; the number of instances to be created is a random integer between 2 and the parameter called ”maxNodes” (line 3) whose value is specified by the user at runtime. The ”name” property of all the created Nodes is set using strings generated by the in-built method, randomString() (line 5). The size of the string to be returned can be configured using (optionally) integers as an argument. The format can also be configured by specifying a string of regular expressions as its argument. The execution of these operations produces a set of model elements that need to be connected together as shown in Figure 5.6a.

Figure 5.6: Unconnected and Connected Model Elements



(a) Unconnected Model Elements

(b) A Complete Graph Model

Lines 7 to 16 specify the linking rule for connecting the generated elements. Each element of type Graph and type Node that is not connected to an incoming Edge (lines 8 and 9) needs to be connected. When a match is found (lines 11 to 14), the Node is connected to the Graph and a new Edge is created that connects the Node to another random Node without any outgoing connection. Figure 5.6b displays a sample generated model.

5.6.2 Eugenia

EMG has also been used to develop a model generator that generates models that satisfy constraints of the Eugenia transformation that was discussed in Section 3.1. Due to the complexity of the constraints, only a subset of the generation rules is discussed here while the complete set of rules is provided in Appendix C. Most of the constraints that do not deal with references to other model elements are satisfied in the creation rule of the appropriate objects while the rest are satisfied using linking rules. The complete set of generation rules contains 6 creation rules, 7 helper EOL operations, 8 linking rules and 2 user-defined parameters ”nodeCount” and ”linkCount” which specifies the approximate number of classes that should be annotated as ”gmf.node” and ”gmf.link” respectively. Listing 5.4 shows a subset of the Eugenia constraints and Listing 5.5 their respective generation rules.

Listing 5.4: Eugenia constraint

```
1 // EVL
2 context EPackage {
3   constraint DiagramIsDefined {
4    check : getDiagramClass().isDefined()
5    message : 'One class must be specified as gmf.diagram'
6   }
7   critique ReferenceLinksAreDefined {
8    guard : self.satisfies('DiagramIsDefined')
9    check : getReferenceLinks().size() > 0 or getLinks().size > 0
10   message : 'No reference links (gmf.link) have been defined'
11  }
12 }
```

```
13 context EClass {
14   guard : self.isLink()
15   constraint LinkSourceIsDefined {
16    check : self.getAnnotationValue('gmf.link', 'source').isDefined()
17    message :'No source defined for link class ' + self.name
18   }
19   constraint LinkSourceExists {
20    guard :self.satisfies('LinkSourceIsDefined')
21    check :self.getReference(self.getAnnotationValue('gmf.link','
          source')).isDefined()
22    message :'No reference named '+ self.getAnnotationValue('gmf.link'
          , 'source')+ ' exists in link class ' + self.name
23   }
24 }
```

Listing 5.5: EMG generation rules

```
1 operation EClass create(){
2   self.name=randomString();
3   self.annotate("gmf.diagram");
4 }
5 $instances Sequence{1,linkCount}
6 operation EClass create(){
7   self.name=randomString();
8   var detail: new Map;
9   detail.put('source', randomString());
10  self.annotate("gmf.link",detail);
11 }
12 pattern linkSource
13   class1:EClass
14   guard: class1.getAnnotationValue("gmf.link","source").isDefined(){
15    onmatch{
16     var r =EReference.createInstance();
17     r.name= class1.getAnnotationValue("gmf.link","source");
18     r.eType= EClass.all.excluding(class1).uniRandom();
19     class1.eStructuralFeatures.add(r);
20    }
21   }
```

The sample Eugenia constraints include four invariants i.e three constraints and one critique, while the EMG program that satisfies these constraints contains two creation rules and one linking rule. The invariants and how they are satisfied by the respective creation and linking rules are discussed below:

1. Constraint DiagramIsDefined: This constraint specifies that exactly one EClass should be annotated as "gmf.diagram". This EClass is designated as the root class in the diagram that will be generated by Eugenia. The constraint is satisfied using the first creation rule (lines 1 to 4) in which an EClass is created and annotated as "gmf.diagram".

2. Critique ReferenceLinksAreDefined: This critique specifies that at least one EClass or a non-containment EReference in the root class should be annotated as ”gmf.link”. These classes or references are represented as links in the diagram. A link in this context is similar to an Edge in the Graph model and it is expected to have a source and a target that it connects together. This critique is satisfied in the second creation rule (lines 5 to 11) that creates a random number of classes between one and the parameter ”linkCount” whose value is specified at runtime.

3. Constraint LinkSourceIsDefined: This constraint specifies that if a class is annotated as ”gmf.link”, then such an annotation must also have an EStringToStringMapEntry with a key of value ”source”. This means that a link in the diagram is expected to have a ”source”. This constraint is also satisfied in the second creation rule by annotating the EClass with an appropriate value.

4. Constraint LinkSourceExists: This constraint verifies that the source of the link is a valid entity in the diagram by specifying that if a class is annotated as ”gmf.link”, then an EReference in the class should be named with the value of the EStringToStringMapEntry key called ”source”. This constraint is satisfied using the linking rule because it needs a reference to another class. Pattern ”linkSource” (lines 12 to 21) satisfies this condition by linking all EClasses annotated as ”gmf.link” which have an EStringToStringMapEntry with key ”source”(guard specification) to a new EReference. If an EClass that fulfils the ”guard” condition is found, a new instance of EReference is created and assigned with the name specified by the constraint ”LinkSourceIsDefined” and a type of any random EClass other than the container EClass.

5.7 Tool Support

In order to make it easy for practical use, a set of development tools that extends existing Epsilon tools is provided as Eclipse plugins. The tools include an editor, a console and a run configuration interface. All the sample graphical representations are provided in Appendix D.

5.7.1 EPL Editor

The EPL editor has been reused because no changes were made to the concrete syntax of the base language. The editor highlights keywords (such as ”operation”, ”pattern”, ”guard”), comments, numbers and strings. Inline markers that highlight places that contain errors are also supported. Figure D.1 in the appendix is a sample EPL editor.

5.7.2 Epsilon Console

This is a common console for all Epsilon languages that provides feedback at runtime. Runtime errors with an hyper link to where they occur in the program are displayed

as texts in the console. The user can also send messages to the console by using in-built EOL operations ”print()” and ”println()”. Figure D.2 in the appendix is a sample Epsilon console.

5.7.3 Run Configuration

A run configuration interface comprising a number of tabs is provided. To generate a Graph model, the ”Source” tab which is shown in Figure D.3, is used to specify the EMG program file (named ”graph.emg”) that contains the model generation rules. A ”seed” can also be specified or randomly generated for repeatability. Figure D.4 shows the ”Models” tab which is used to specify features (such as name and metamodel) of the model to be generated. If the ”add” button is clicked, a new interface for specifying these features is displayed. As seen in Figure D.5, the name of the model is ”graph”, its metamodel is ”graph.ecore” and it will be stored in a file named ”graph.graph”. As the file does not exist before the execution of the program, we do not select the ”Read on load” checkbox, however we expect the model to be stored in a file after its execution. User-defined parameters such as ”nodeCount” are specified in the ”Parameters” tab as shown in Figure D.6. All the figures are provided in the appendix.

5.8 Chapter Summary

This chapter has identified recurring tasks in model generation processes and introduced a novel 2-step semi-automated approach to model generation. The Epsilon Model Generation (EMG) framework that simplifies the development of model generators and implements this approach has been presented. The framework is a semantic extension to the Epsilon Pattern Language (EPL) and provides first class support for model generation tasks. A discussion on how the framework can be used to produce models that can exhibit desired characteristics such as repeatability and randomness was also provided. Sample model generators were also illustrated and the available supporting tools were presented.

6 Evaluation

This thesis has introduced a novel approach towards generation of models. Based on this approach, a framework was developed to simplify the development of model generators that can produce repeatable and random models. This chapter evaluates to what extent the research hypothesis is valid and how well the framework satisfies the research objectives stated in Section 4.2. Section 6.1 examines whether EMG can be used to generate models that conform to complex constraints for which fully-automated model generators fail. Section 6.2 examines the robustness of the built-in random functions provided by EMG and the ability of the framework to reproduce generated models. Section 6.3 compares EMG with a general purpose language and in Section 6.4, a comparison with other frameworks that may be able to generate similar models is conducted. Section 6.5 compares the semi-automated approach to existing fully-automated approaches while Section 6.6 discusses the shortcomings of this work.

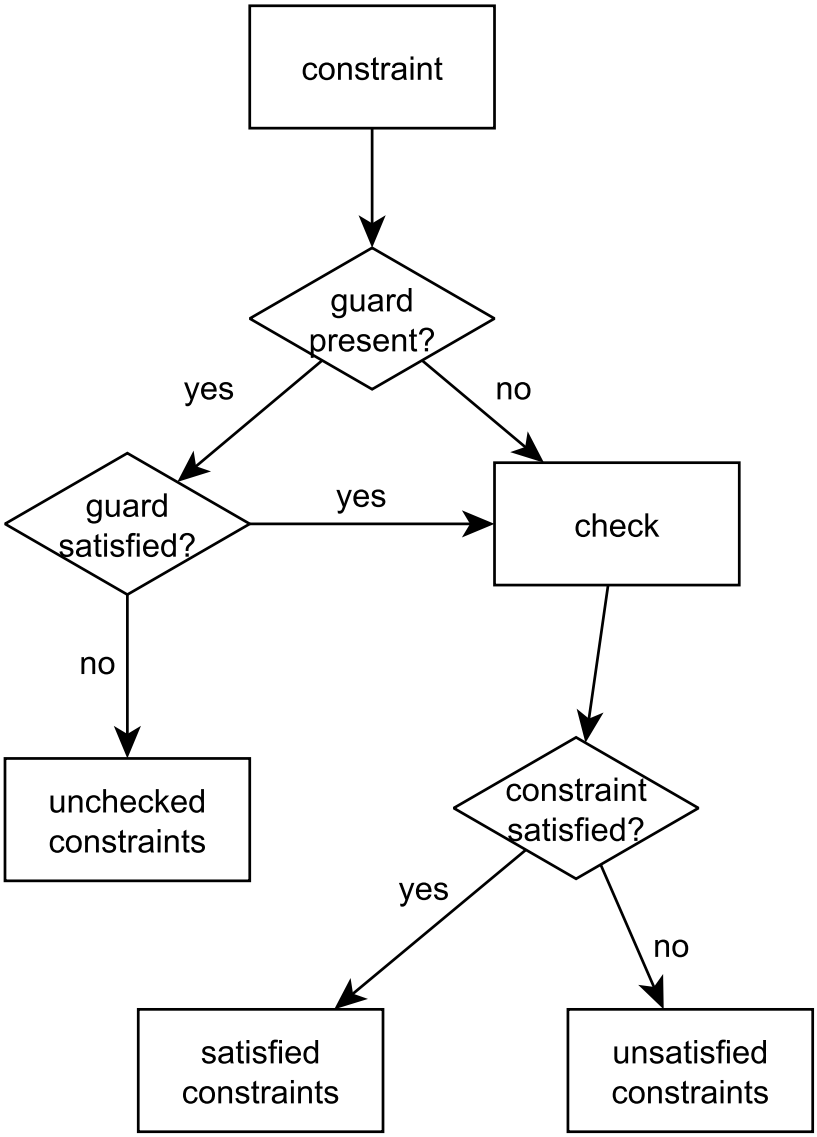
6.1 Generation of Models with Complex Constraints

In Section 5.6.2, a model generator that can produce models that satisfy Eugenia-specific constraints was developed using EMG. This section assesses whether the generated models fully exercise and satisfy the Eugenia constraints using a test program. The test program creates instances of EmgModules that contain the generation rules and EvlModules that contain the Eugenia constraints. The models generated by the EmgModule are validated by the EvlModule at runtime.

Figure 6.1 shows how invariants are evaluated in EVL based on the explanations in [7]. The invariants can only be evaluated if the ”guard” condition is satisfied or if the invariants contain no ”guard” conditions. If the guard is not satisfied, the constraint is unchecked and it cannot be ascertained if the generator can produce models that can satisfy that particular constraint or not. Hence we analysed the generated models based on not only when the constraints are satisfied but also when the ”guard” condition was satisfied and the number of times each constraint was evaluated.

Table 6.1 presents the results of the analysis on models generated after 100 model generation cycles for three groups of models containing 10, 50 and 100 EClasses which gives a total of three hundred models. In all generated models, no unsatisfied constraint was observed i.e there was no constraint that meets the ”guard” condition and was unsatisfied. This gives a high probability that our generator can truly generate valid models that satisfy the constraints. There was no constraints that was not examined

Figure 6.1: Evaluation of EVL constraints



(checked) at least once in any group of models. Figure 6.2 shows the total number of models where each constraint is checked in the group of models containing about 100 EClasses each. Also each model has between 16 and 24 constraints checked while each of the constraints was checked in at least 28 out of the 100 models. The total number of constraints in Eugenia is 26 and a detailed explanation of each of the constraints is provided in appendix A.

Figure 6.2: Number of Times Each Constraint is Satisfied

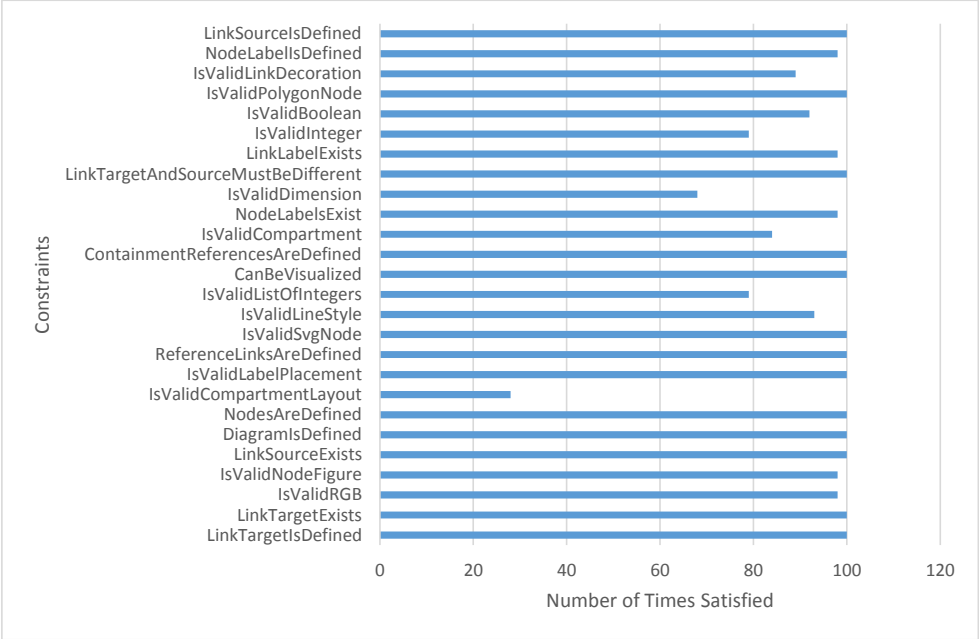


Figure 6.3: Number of Constraints Satisfied in Each Model

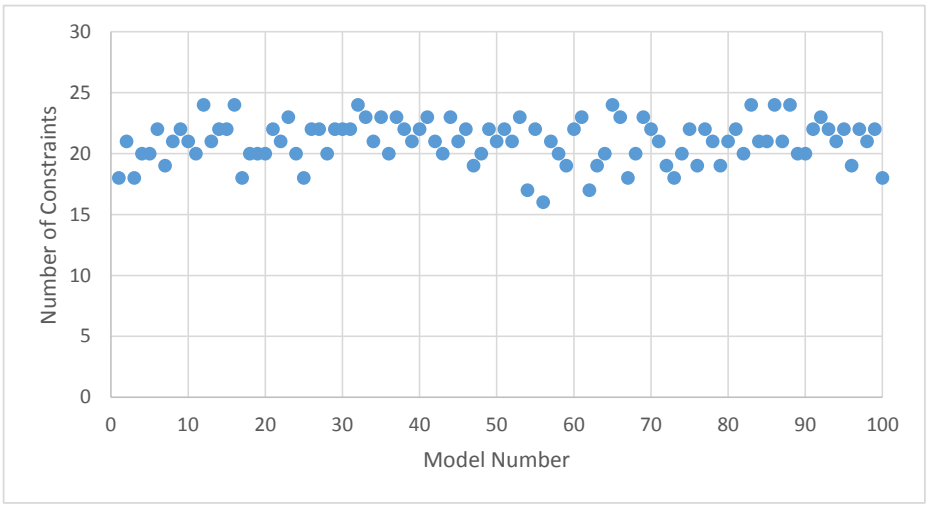


Table 6.1: Analysis of Models Generated

Approx. EClass	Unsatisfied Constraints	Constraints Satisfied Every Time	Min models "checked" for Any Constraint
10	0	11	18
50	0	12	27
100	0	13	28

6.2 Randomness and Repeatability

100 different models with about 100 EClasses each were generated to test for the robustness of the random functions using the default uniform distribution. Our results show great diversity in the number of models generated as shown in Figure 6.4. Furthermore, the range of values was also random as it was observed that for the EClass annotated as "gmf.diagram", no two EClasses had the same name in the 100 generated models.

Another set of 100 models were further generated using a specified seed in order to assess the ability of the framework to reproduce generated models. Exactly the same model with the same structure and values assigned to model elements was generated throughout these cycles.

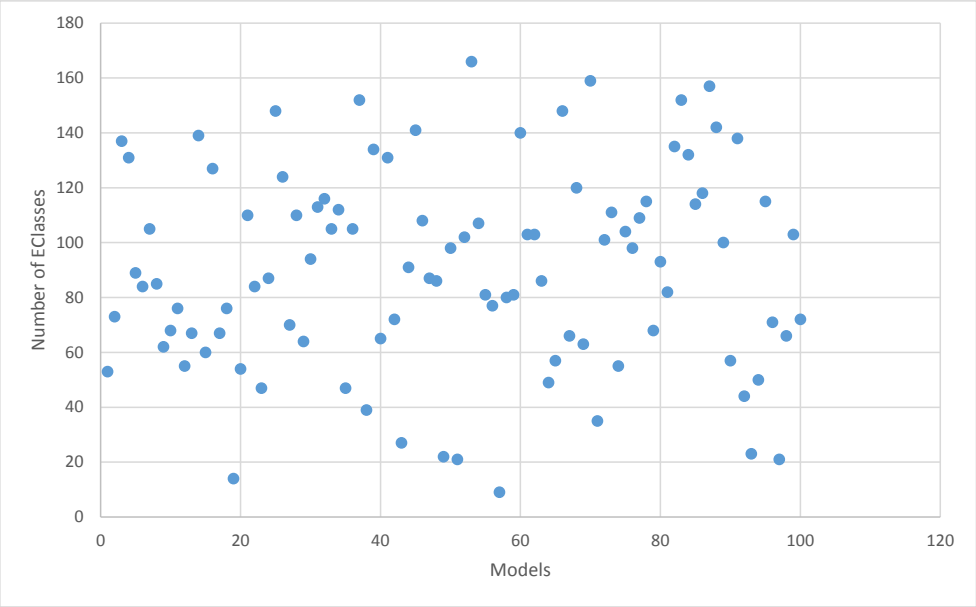
6.3 Comparison with a General Purpose Language

This section compares a model generator developed in EMG with a similar bespoke generator constructed using EOL in terms of conciseness, performance and ability to reproduce generated models. EOL has been chosen as an example of an imperative model-oriented programming language. Other languages such as QVTo or Kermeta could also have been used for this purpose.

6.3.1 Case Study

Our case study is a real-life railway system developed by domain experts [88]. The railway is composed of Routes, Signals, Switches, SwitchPositions, (abstract) TrackElements, Segments and Sensors. A Route is defined by a set of Sensors. Sensors are associated with TrackElements which are either Segments (with a specific length) or Switches. A Route can also be associated to SwitchPositions which describe the required state of a Switch belonging to the route. Different Routes can specify different states for a specific Switch. Figure 6.5 illustrates the metamodel of the system in Ecore [89] while Figure 6.5(b) depicts the containment hierarchy among the elements in the metamodel. The system is also expected to satisfy the following constraints.

Figure 6.4: Number of EClasses Generated for 100 Models



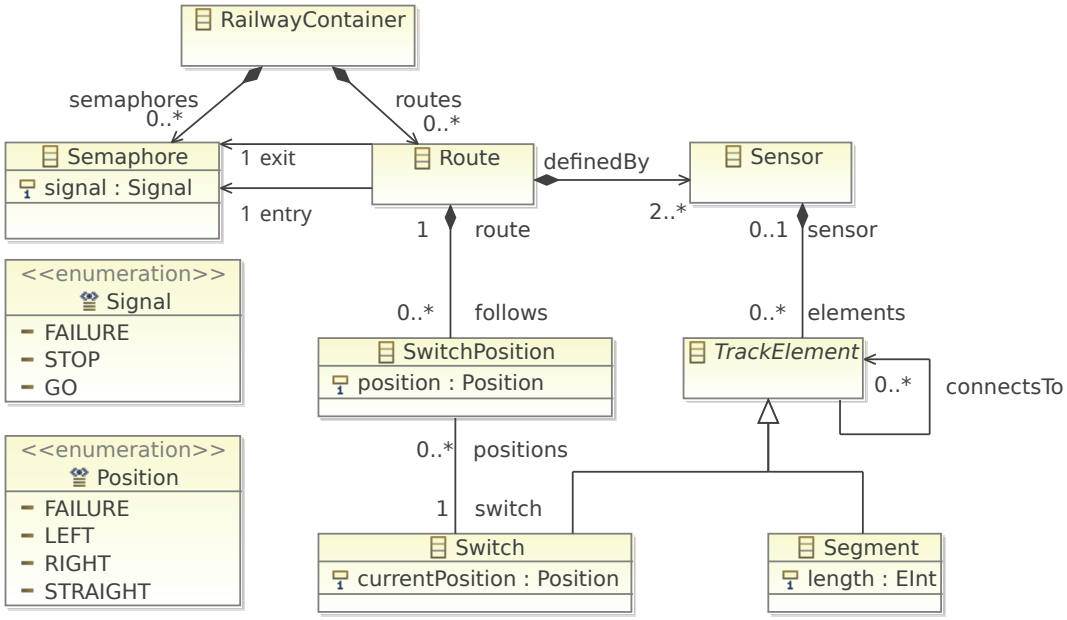
1. Semaphore-Neighbour constraint: Routes connected through Sensor and TrackElement must belong to the same Semaphore.
2. PosLength: the length of a segment should be positive.
3. SwitchSensor: every switch must have at least one sensor connected to it.
4. SwitchSet: the entry semaphore of a route should only show GO if all the switches associated with the route are in the position prescribed by the route.

Figure 6.6(a) and (b) is a visual representation of the Semaphore-Neighbour and Switchset constraints respectively. Listing 1 and Listing 2 in Appendix E are sample EMG and EOL programs written by me that can be used to generate appropriate models respectively.

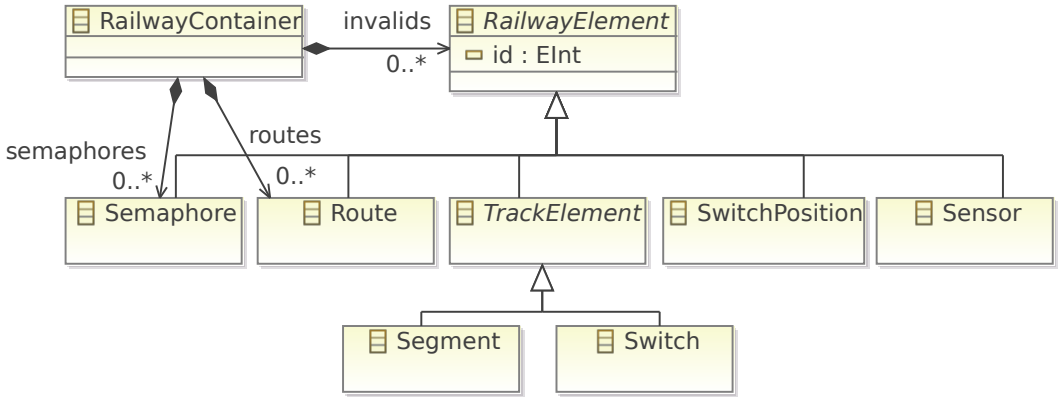
6.3.2 Analysis

EMG is significantly less verbose than EOL because it needs a lesser amount of code to generate similar models. A total of 65 lines of code were used in EMG while 100

Figure 6.5: Railway Metamodel



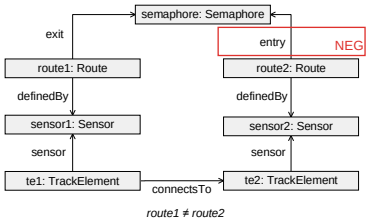
(a) Containment Hierachy and References



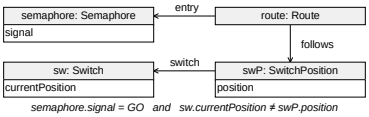
(b) Supertype Relationships

lines of code consisting of 19 ”for” loops and 13 ”if” statements were used to generate similar models in EOL. Furthermore, EOL does not provide capabilities for reproducing generated models and supports only uniform random distribution. Both EMG and EOL produce similar models.

Figure 6.6: Added Constraints



(a) Semaphore-Neighbour Constraint



(b) Switchset Constraint

Figure 6.7: Execution times for EOL and EMG

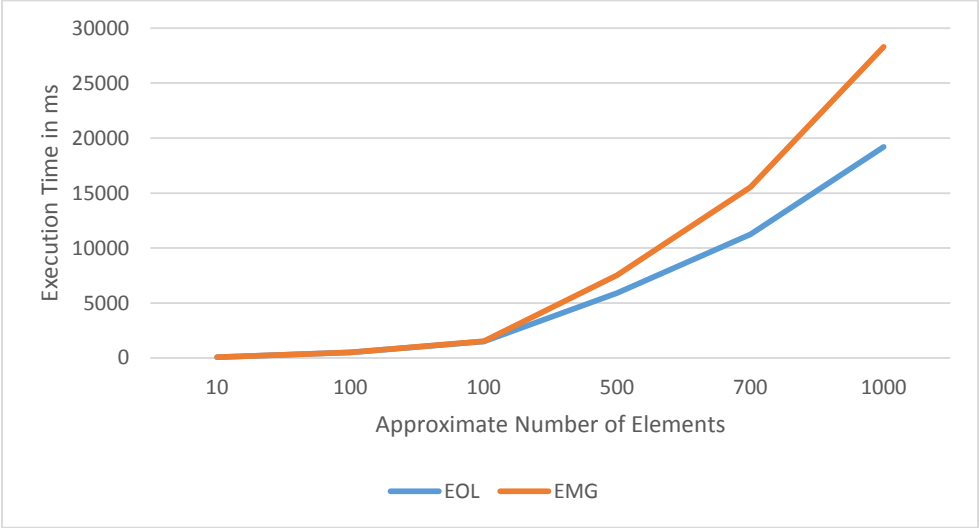


Figure 6.7 summarizes the execution times of EOL and EMG in generating Railway models with different number of model elements. The results show that EOL currently performs better than EMG. This is because EMG extends EPL and its internal execution mechanism has not been optimized for model generation. This limitation could not be addressed in this work due to time constraints and has been identified as a topic for future research.

6.4 Comparison with other Generators

The only framework providing similar functionality to EMG is RandomEMF [67], a framework for generating large random models that can be used for benchmarking. The framework also adopts a semi-automated approach to model generation, similar to the one used in EMG. RandomEMF's generation rules are specified with Rcore, a language developed using Xtext[1]. RandomEMF was developed about the same time with this work.

In EMG, all the required model elements are first generated before linking them together but in RandomEMF, the linking is done as soon as an element is created which means the elements generated later are not considered for the linking operation. Listing 6.1 is a RandomEMF program that generates Graph models. The number of Edges is also manually determined which may be inadequate or excessive for connecting the Nodes. A Graph model was used as a case study because the RandomEMF code for generating Graph models was produced by the developer of RandomEMF. This is important to minimize bias due to our limited knowledge of RandomEMF.

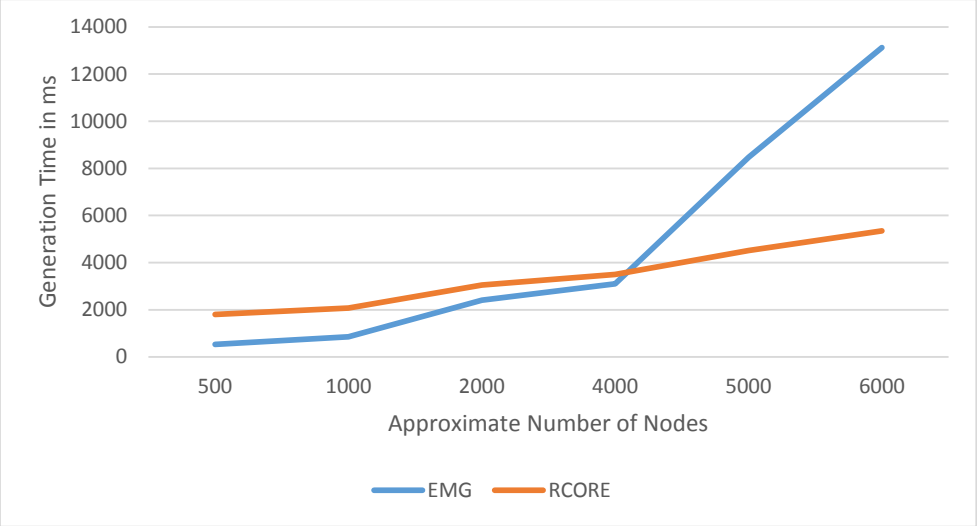
Listing 6.1: Rcore Program to Generate Graph Model

```
1  package de.hub.rcore.graph
2  import static de.hub.randomemf.runtime.Random.*
3  import static de.hub.rcore.graph.RandomGraphUtil.*
4  generator RandomGraph(int nodeCount, int edgeCount) for graph in "
       platform:/resource/de.hub.rcore.graph/model/graph.ecore" {
5  root: Graph ->
6    nodes += node#nodeCount
7    edges += edge#edgeCount
8  ;
9  node: Node ->
10   name := LatinCamel(Normal(4,2)).toFirstLower
11 ;
12 edge: Edge ->
13   name := LatinCamel(Normal(4,2)).toFirstLower
14   source := @(model.nodes.get(Uniform(0,model.nodes.size)))
15   target := @(reject(self.source)
16   [model.nodes.get(Uniform(0,model.nodes.size))])
17 ;
18 }
```

Similar number of codes were used by both EMG and RandomEMF to generate the Graph models. The quality of the generated models were also similar. Figure 6.8 shows that RandomEMF is faster for generating models with large number of model elements. This is largely because EMG is an interpreted language while Rcore compiles down to Java.

[1]https://eclipse.org/Xtext/

Figure 6.8: Execution Times for EMG and RandomEMF



6.5 Comparison with Other Approaches

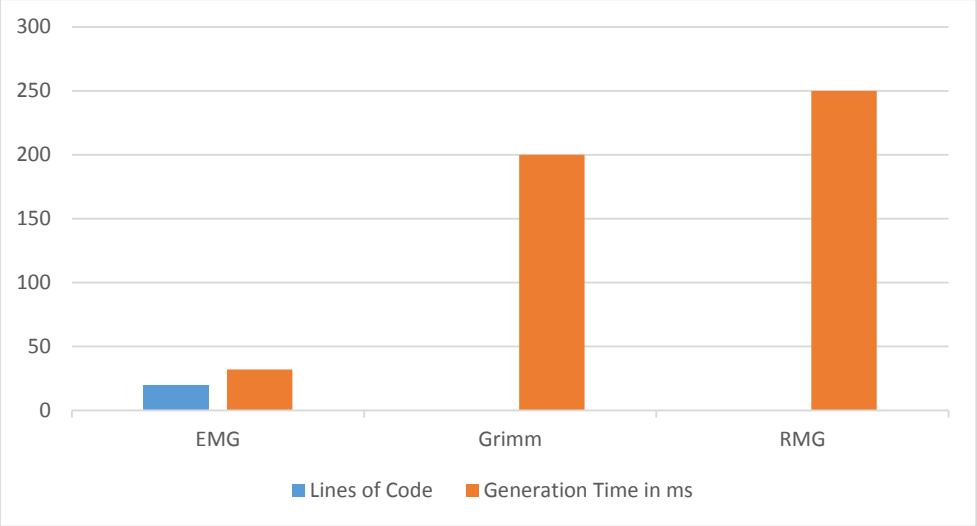
This section compares the semi-automated generation approach adopted in this thesis with similar approaches listed in Section 2.4. The semi-automated approach is similar to the configuration approach. However instead of automatically generating a configuration model, the developer manually constructs generation rules to guide the generation process. Using the same criteria listed in Section 2.4, the following are observed:

1. Validity: This approach can be used to generate valid models as discussed in Section 6.1. However, the responsibility for ensuring that generated models conform to the Ecore metamodel and satisfy the required constraints lies with the developer of the model generation rules.

2. Adequacy: The approach is adequate i.e it can be used to generate models that satisfy arbitrarily complex constraints.

3. Scalability: The ability of this approach to efficiently generate models with a large number of model elements cannot be established for now.

Figure 6.9: Performance Evaluation for EMG, Grimm and RMG



4. Flexibility: This approach can be used to develop flexible model generators that are easy to parameterize. Section 5.4.6 and the running examples in Section 5.6 shows that EMG is easy to parameterize.

Although the scalability of EMG cannot be established for now, its performance has been compared with tools that are developed using the Constraint Satisfaction approach. Tools that were developed based on Model Fragment and Tree approaches are not currently available while the execution time of MM2GRAGRA [70], the only available tool that implements the Configuration approach, cannot be estimated. Only the execution times of Grimm [55] and RMG [71] can be programatically estimated. Furthermore, it was not possible to configure the number of model elements in the generated model with these tools, hence only models with minimal elements were produced. Averagely, EMG was able to generate a Graph model in 32ms while Grimm and RMG execution times were 200ms and 250ms respectively. Both Grimm and RMG were fully automated tools, hence no code was required to produce a Graph model, while EMG uses 20 lines of code to generate a similar model. Figure 6.9 summarizes the results of this exercise.

Table 6.2: Comparison with other Approaches

Approaches	Valid	Adequate	Scalable	Flexible
Semi-Automated	Yes	Yes	?	Yes
Constraint Satisfaction	Yes	?	No	Yes
Model Fragmentation	?	Yes	No	Yes
Configuration	No	Yes	Yes	No
Tree	No	No	Yes	No

Table 6.2 summarizes the comparison of this approach to fully automated approaches. In the semi-automated approach implemented by EMG, a question mark (?) denotes that the criterion is yet to be proven. In other approaches, it denotes that such criterion cannot be verified from the literature for the specific approach.

6.6 Shortcomings

A technical shortcoming of EMG is its performance for generating models with large numbers of model elements, hence it may not be ideal for generating models for benchmarking. To address this limitation, optimisation of the EMG interpreter and/or compilation of EMG programs to a more performant representation (e.g. Java) are directions for future exploration.

6.7 Threats to Validity

EMG has been used to develop only a small number of model generators. Hence, it may be possible that EMG is not more concise than EOL for developing any generator.

6.8 Chapter Summary

This chapter has shown that EMG can be used to produce random and repeatable models that satisfy complex constraints. It has also been shown that the framework can be used to develop more concise model generators than constructing them with a general purpose language thus validating the first part of the research hypothesis. A shortcoming in terms of processing speed for large models has been identified and recommendations that can help alleviate this shortcoming have been provided.

7 Conclusion and Future Work

7.1 Review Findings

Chapter 2 and Chapter 3 of this thesis provided a review of related work in automated model generation. We've identified four main approaches and several tools implemented using this approaches. We were also able to identify that:

1. None of the existing tools were able to generate models that satisfy arbitrarily constraints.
2. Existing approaches do not deal with repeatability of the models generated and none of the tools was able to reproduce generated models.
3. Existing tools attempt to generate instances of all the classes in the metamodel even if not all the classes are necessary.

7.2 Research Hypothesis

The research challenges identified in our review of related work was tackled with respect to the following hypothesis originally stated in Section 4.2:

A dedicated language for model generation can be used to develop more concise, configurably random and reproducible model generators that perform better than generators constructed with a general purpose language without sacrificing expressiveness and readability.

The objectives of this research were to:

1. Identify recurring patterns during the model generation process.
2. Encode the identified patterns in the syntax of a dedicated language.
3. Evaluate the validity of the hypothesis beginning with conciseness and ability to reproduce models.

7.3 Prototype Solution

With respect to the research objectives of this thesis, Section 5.1 identified recurring patterns in a bespoke model generation process. Section 5.2 introduced a novel 2-step semi-automated approach towards model generation and presented a framework with

a language based on this approach. The framework named Epsilon Model Generator (EMG), is a semantic extension to Epsilon Pattern Language (EPL) while its language can be used for specifying model generation rules. The syntax of the language also encoded the identified patterns in Section 5.1. We were able to demonstrate how the framework can be used to generate models that exhibit properties such as randomness, repeatability and flexibility.

7.4 Evaluation Results

This section states how the research objectives have been met and to what extent the research hypothesis has been validated. The objectives of this thesis were to:

1. Identify recurring patterns during the model generation process.
2. Encode the identified patterns in the syntax of a dedicated language.
3. Evaluate the validity of the hypothesis.

Chapter 5 identified recurring patterns in a bespoke model generation process and encoded the identified patterns in the syntax of a dedicated language named EMG thereby satisfying the first and second objectives of this research. The evaluation of the research hypothesis has been conducted in Chapter 6 thus achieving the third objective.

The research hypothesis of this thesis, which was originally presented in Section 4.2, states that:

*A dedicated language for model generation can be used to develop **more concise**, **configurably random** and **reproducible** model generators that **perform better** than generators constructed with a general purpose language **without sacrificing expressiveness and readability**.*

The extent to which this hypothesis has been validated is stated below.

1. Conciseness: We have shown in Section 6.3 that EMG is more concise than a general purpose language.
2. Configurable randomness: EMG provides built-in support for parametric random operations which are necessary in order to generate models of different sizes using the same EMG specification.
3. Reproducibility: EMG is able to reproduce generated models by using the same seed as shown in Section 6.2.
4. Performance: Performance was found to be a weakness of EMG in comparison with EOL. Additional work (e.g. optimisation or compilation instead of interpretation) is needed to improve EMG's performance.

5. Expressiveness: It has been demonstrated that EMG can be used to produce models that satisfy complex constraints.

6. Readability: Section 6.3 shows that EMG can be used to construct model generators with fewer low-level constructs such as ”for” and ”if” statements than a general purpose language, which arguably enhances readability.

In summary, the research objectives of this thesis has been met and the first part of the research hypothesis has been validated. The second part of the hypothesis could not be validated and is proposed as a possible direction for future work.

7.5 Future Work

Some directions for improving on this work have been identified. In terms of speed, it may be necessary to optimise the EMG interpreter and/or compile EMG programs to a more performant representation (e.g. Java) in order to reduce execution times for models with large model elements. In order to minimize the manual process of writing EMG programs, automated transformation of EVL constraints to EMG rules may be considered.

A Eugenia Constraints

This section presents the Eugenia constraints [1] discussed in Section 3.1 and provides a short explanation of each of them. The Eugenia constraints apply to Ecore models and are written in EVL. A typical EVL constraint contains a name, a guard block that specifies the pre-conditions to be satisfied before the constraint can be executed, a check block that specifies the success condition and a message block that contains information to be sent to the user if the constraint is not satisfied. An EVL constraint may also have a context type which means that the constraint would be evaluated for only objects of that type. In EVL, a Critique is used to highlight desired but optional properties a model should have. A Critique is similar to a Constraint except that an unsatisfied Constraint invalidates the model while an unsatisfied Critique only produces a warning. The imported EcoreUtil.eol file is available online [2]

```
1  import 'ECoreUtil.eol';
2
3  context EPackage {
4
5   constraint DiagramIsDefined {
6
7    check : getDiagramClass().isDefined()
8
9    message : 'One class must be specified as gmf.diagram'
10
11  }
12
13  constraint ContainmentReferencesAreDefined {
14
15   guard : self.satisfies('DiagramIsDefined')
16
17   check : getDiagramClass().getContainmentReferences().size()
         > 0
18
19   message : 'Diagram class ' + getDiagramClass().name + '
       must define '
20   + ' at least one containment reference'
21
22  }
23
24  constraint NodesAreDefined {
25
26   guard : self.satisfies('DiagramIsDefined')
27
28   check : getNodes().size() > 0
```

[1] https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/plain/plugins/org.eclipse.epsilon.eugenia/transformations/ECore2GMF.eol

[2] https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/plain/plugins/org.eclipse.epsilon.eugenia/transformations/ECoreUtil.eol

```
29
30          message : 'No nodes (gmf.node) have been defined'
31
32         }
33
34         critique ReferenceLinksAreDefined {
35
36          guard : self.satisfies('DiagramIsDefined')
37
38          check : getReferenceLinks().size() > 0 or getLinks().size >
                0
39
40          message : 'No reference links (gmf.link) have been defined'
41         }
42
43        }
44
45        context EClass {
46
47         guard : self.isNode()
48
49         constraint IsValidSvgNode {
50
51          check : (self.getAnnotationValue('gmf.node', 'figure') = "
                svg") implies
52          (self.getAnnotationValue('gmf.node', 'svg.uri').isDefined()
                )
53
54          message : "No svg.uri provided for SVG figure " + self.name
55
56         }
57
58         constraint IsValidPolygonNode {
59
60          check : (self.getAnnotationValue('gmf.node', 'figure') = "
                polygon") implies
61          (self.getAnnotationValue('gmf.node', 'polygon.x').isDefined
                () and self.getAnnotationValue('gmf.node', 'polygon.y').
                isDefined())
62
63          message : "No polygon x/y coordinates provided for polygon
                figure " + self.name
64
65         }
66
67         constraint NodeLabelIsDefined {
68          guard : self.getAnnotationValue('gmf.node', 'label.
                placement') <> "none"
69
```

```
70          check : self.getAnnotationValue('gmf.node', 'label').
                isDefined()
71
72          message : 'No label defined for class ' + self.name
73         }
74
75         constraint NodeLabelsExist {
76
77          guard : self.satisfies('NodeLabelIsDefined')
78
79          check {
80          var missing : Sequence(String);
81          for (label : String in self.getAnnotationValue('gmf.node',
                'label').split(',').collect(s|s.trim())){
82          if (not self.getAttribute(label).isDefined()) {
83           missing.add(label);
84          }
85          }
86          return missing.size() = 0;
87         }
88
89          message : 'Label attribute(s) ' + missing.concat(', ')
90          + ' do(es) not exist in class ' + self.name
91
92         }
93
94        }
95
96        context EAnnotation {
97
98         constraint IsValidCompartment {
99
100          guard : self.source = "gmf.compartment"
101
102          check : self.eContainer().isTypeOf(EReference) and self.
                eContainer().containment = true
103
104          message : "EReference " + self.eContainer().name + " is not
                 a containment reference"
105
106         }
107
108        }
109
110        /*
111        * Validation rules for annoations details
112        */
113        context EStringToStringMapEntry {
114
```

```
115         critique IsValidLinkDecoration {
116
117         guard : self.is('gmf.link', 'source.decoration') or
118         self.is('gmf.link', 'target.decoration')
119
120         check {
121         var values := Sequence{'none', 'arrow', 'rhomb', '
                filledrhomb',
122         'square', 'filledsquare', 'closedarrow', 'filledclosedarrow'
                };
123
124         return self.value.isWithinValuesOrLooksLikeJavaClassName(
                values);
125
126         }
127
128         message : 'The value of ' + self.toEmfatic() + ' must be one
                of: ' + values.concat(', ') +
129         " or a fully-qualified Java class name"
130         }
131
132         critique IsValidNodeFigure {
133
134          guard : self.is('gmf.node', 'figure')
135
136          check {
137          var values := Sequence{'rectangle', 'ellipse', 'rounded', '
                svg', 'polygon'};
138
139          return self.value.isWithinValuesOrLooksLikeJavaClassName(
                values);
140
141          }
142
143          message : 'The value of ' + self.toEmfatic() + ' must be
                one of: ' + values.concat(', ') + " or a fully-qualified
                 Java class name"
144         }
145
146         constraint IsValidLabelPlacement {
147
148          guard : self.is('gmf.node', 'label.placement')
149
150          check {
151          var values := Sequence{'internal', 'external', 'none'};
152          return values.includes(self.value);
153          }
154
```

```
155         message : 'The value of ' + self.toEmfatic() + ' must be
                one of: ' + values.concat(', ')
156         }
157
158       constraint IsValidInteger {
159         guard : self.is('gmf.node', 'border.width') or
160         self.is('gmf.node', 'margin')
161
162         check : self.value.isInteger()
163
164         message : 'The value of ' + self.key + " is not a valid
                integer"
165       }
166
167       constraint IsValidListOfIntegers {
168         guard : self.is('gmf.node', 'polygon.x') or
169         self.is('gmf.node', 'polygon.y')
170
171         check : self.value.matches("(\\s*\\d+)+")
172
173         message : 'The value of ' + self.toEmfatic() + " is not a
                valid list of integers"
174       }
175
176       constraint IsValidDimension {
177         guard : self.is('gmf.node', 'size')
178
179         check : self.value.matches("\\s*\\d+,\\s*\\d+\\s*")
180
181         message : 'The value of ' + self.toEmfatic() + " is not a
                valid dimension"
182       }
183
184       constraint IsValidRGB {
185         guard : self.is('gmf.node', 'border.color') or
186         self.is('gmf.node', 'color') or
187         self.is('gmf.link', 'color')
188
189         check : self.value.matches("\\s*\\d+,\\s*\\d+,\\s*\\d+\\s*"
                )
190
191         message : 'The value of ' + self.toEmfatic() + " is not a
                valid RGB color"
192       }
193
194       constraint IsValidBoolean {
195
196         guard : self.is('gmf.diagram', 'rcp') or
197         self.is('gmf.node', 'label.icon') or
```

```
198          self.is('gmf.node', 'label.readOnly') or
199          self.is('gmf.node', 'phantom') or
200          self.is('gmf.node', 'resizable') or
201          self.is('gmf.compartment', 'collapsible') or
202          self.is('gmf.label', 'label.readOnly')
203
204          check {
205          var values := Sequence{'true', 'false'};
206          return values.includes(self.value);
207          }
208
209          message : 'The value of ' + self.toEmfatic() + ' must be
                 one of: ' + values.concat(', ')
210        }
211
212        constraint IsValidLineStyle {
213
214          guard : self.is('gmf.node', 'border.style') or
215          self.is('gmf.link', 'style')
216
217          check {
218          var values := Sequence{'dot', 'dash', 'solid'};
219          return values.includes(self.value);
220          }
221
222          message : "The value of " +  self.toEmfatic() + " must be
                 one of: " + values.concat(', ')
223        }
224
225        constraint IsValidCompartmentLayout {
226
227          guard : self.key = 'layout'
228
229          check {
230          var values := Sequence{'list', 'free'};
231          return values.includes(self.value);
232          }
233
234          message : 'The layout of the ' + self.eContainer().
                 eContainer().name +
235          ' compartment must be one of: ' + values.concat(', ')
236          }
237        }
238
239      context EClass {
240
241        guard : self.isLink()
242
243        constraint LinkLabelExists {
```

```
244
245          guard : self.getAnnotationValue('gmf.link', 'label').
                 isDefined()
246
247          check {
248          var missing : Sequence(String);
249          for (label : String in self.getAnnotationValue('gmf.link',
                 'label').split(',').collect(s|s.trim())){
250           if (not self.getAttribute(label).isDefined()) {
251           missing.add(label);
252           }
253          }
254          return missing.size() = 0;
255          }
256
257          message : 'Label attribute(s) ' + missing.concat(', ')
258          + ' does not exist in link class ' + self.name
259
260        }
261
262
263        constraint LinkSourceIsDefined {
264
265          check : self.getAnnotationValue('gmf.link', 'source').
                 isDefined()
266
267          message : 'No source defined for link class ' + self.name
268
269        }
270
271        constraint LinkSourceExists {
272
273          guard : self.satisfies('LinkSourceIsDefined')
274
275          check : self.getReference(self.getAnnotationValue('gmf.link
                 ', 'source')).isDefined()
276
277          message : 'No reference named ' + self.getAnnotationValue('
                 gmf.link', 'source')
278          + ' exists in link class ' + self.name
279
280        }
281
282        constraint LinkTargetIsDefined {
283
284          check : self.getAnnotationValue('gmf.link', 'target').
                 isDefined()
285
286          message : 'No target defined for link class ' + self.name
```

```
287
288         }
289
290         constraint LinkTargetExists {
291
292          guard : self.satisfies('LinkTargetIsDefined')
293
294          check : self.getReference(self.getAnnotationValue('gmf.link
                ', 'target')).isDefined()
295
296          message : 'No reference named ' + self.getAnnotationValue('
                gmf.link', 'target')
297          + ' exists in link class ' + self.name
298
299         }
300
301         constraint LinkTargetAndSourceMustBeDifferent {
302
303          guard : self.satisfies('LinkSourceExists') and self.
                satisfies('LinkTargetExists')
304
305          check : self.getAnnotationValue('gmf.link', 'source') <>
                self.getAnnotationValue('gmf.link', 'target')
306
307          message : 'Source and target attributes must be different
                in link class ' + self.name
308
309         }
310
311         critique CanBeVisualized {
312
313          guard : getDiagramClass().isDefined()
314
315          check : getDiagramContainmentReference(self).isDefined()
316
317          message : 'Cannot generate link for class ' + self.name +
318          ' because it cannot be contained in any containment
                reference' + ' of diagram root ' + getDiagramClass().
                name
319         }
320        }
321
322        operation String isWithinValuesOrLooksLikeJavaClassName(
             values : Sequence) {
323         return values.includes(self) or self.indexOf('.') > -1;
324        }
325
326        operation EStringToStringMapEntry is(annotation : String, key
             : String) {
```

```
327         return self.eContainer().source = annotation and self.key =
                key;
328       }
329
330       operation ECore!EClass getAttribute(name : String) {
331         return self.eAllStructuralFeatures.selectOne(sf:ECore!
                EAttribute|sf.name = name);
332       }
333
334       operation ECore!EClass getReference(name : String) {
335         return self.eAllStructuralFeatures.selectOne(sf:ECore!
                EReference|sf.name = name);
336       }
337
338       operation EStringToStringMapEntry toEmfatic() {
339         var s = "@" + self.eContainer.source + "(" + self.key + ")";
340         if (self.eContainer().isKindOf(EStructuralFeature)) {
341           s = s + " of " + self.eContainer().eContainer().eContainer
                  ().name + "." +
342           self.eContainer().eContainer().name;
343         }
344         else {
345           s = s + " of " + self.eContainer().eContainer().name;
346         }
347         return s;
348       }
349
350       operation Any getTopEPackage() {
351         if (self.eContainer().isDefined()) {
352           return self.eContainer().getTopEPackage();
353         }
354         else {
355           return self;
356         }
357       }
```

Constraint "DiagramIsDefined" specifies that one class should be annotated as "gmf.diagram" as defined in the EOL Operation "getDiagramClass()". Constraint "ContainmentReferencesAreDefined" states that the class annotated as "gmf.diagram" should have at least one containment reference. Constraint "NodesAreDefined" specifies that at least one class should be annotated as "gmf.node" while Critique "ReferenceLinksAreDefined" states that at least one class should be annotated as "gmf.link".

The following constraints are also specified for a class that has been annotated as "gmf.node": Constraint "IsValidSvgNode" specifies that if the "gmf.node" annotation contains a key named "figure" with a value of "svg", then a key named "svg.uri" must be defined. Constraint "IsValidPolygonNode" specifies that if the "gmf.node" annotation contains a key named "figure" with a value of "svg", then two keys named "ploygon.x"

and ”polygon.y” must be defined. Constraint ”NodeLabelIsDefined” specifies that if a key named ”label.placement” has a value of ”none”, then a key named ”label” must be defined. Constraint ”NodeLabelsExist” states that if Constraint ”NodeLabelIsDefined” is satisfied, then the class annotated with gmf.node must have ”attributes” specified by the value of the key named ”label”.

Constraint ”IsValidCompartment” specifies that if a class is annotated as ”gmf.compartment”, then the class must be a containment reference. Critique ”IsValidLinkDecoration” states that if a ”gmf.link” annotation has a key named ”source.decoration” or ”target.decoration”, then its value must be one of the values stated in the Critique’s check block. Critique ”IsValidNodeFigure ” states if a ”gmf.node” annotation has a key named ”figure”, then its value must be one of the values stated in the Critique’s check block. Constraint ”IsValidLabelPlacement” specifies if a ”gmf.node” annotation has a key named ”label.placement”, then its value must be one of the values stated in the Constraint’s check block. Constraint ”IsValidInteger” specifies that the value of the ”gmf.node” annotation’s key named ”border.width” or ”margin” must be a valid integer. Constraint ”IsValidListOfIntegers” checks for valid list of integers while Constraint ”IsValidDimension” checks for valid dimensions for associated ”gmf.node” annotations. Constraint ”IsValidRGB” specifies a valid color, Constraint ”IsValidBoolean” specifies a valid Boolean value, Constraint ”IsValidLineStyle” checks if the style of the line to be drawn is valid while Constraint ”IsValidCompartmentLayout” validates the layout of associated annotations.

If a class has been annotated as ”gmf.link”, the following constraints are expected to be satisfied: Constraint ”LinkLabelExists” states that if a key named ”label” is defined, then its value (separated by commas) should be attributes of the class. Constraint ”LinkSourceIsDefined” and Constraint ”LinkSourceExists” specifies that a key named ”source” should be defined and a value should be assigned to it while Constraint ”LinkTargetIsDefined” and Constraint ”LinkTargetExists” specifies that a key named ”target” should be defined and a value should be assigned to it. Constraint ”LinkTargetAndSourceMustBeDifferent” states that the values of the keys named ”source” and ”target” must be different from each other while Constraint ”CanBeVisualized” validates that this class (annotated as ”gmf.link”) is a containment reference for the class annotated as ”gmf.diagram”. Some of the EOL operations that were used in some of the constraints were also defined in the code. Other EOL operations were defined in ”EcoreUtil.eol” that was imported into the EVL code in line 1.

B OCL Translation for Eugenia Constraints

This is a direct OCL translation of the Eugenia constraints (originally written in EVL). This translation was needed in order to assess existing fully-automated tools that do not support EVL. The translation was done by the author and its faithfulness has been verified through testing. The assessment of existing fully-automated tools was discussed in Section 3.2.

```
1  import 'http://www.eclipse.org/emf/2002/Ecore'
2  package ecore
3
4  context EPackage
5
6   def: isDiagramDefined():EClass = self.eClassifiers-> select(
        c | c.oclIsTypeOf(EClass) and c->exists(a| a.
       eAnnotations->select(s|s.source = 'gmf.diagram') ))
7
8   def: NodesAreDefined():EBoolean= self.eClassifiers->select (
       a|a.eAnnotations-> exists (s|s.source = 'gmf.node'))
9
10  def: ReferenceLinksAreDefined():EBoolean= self.eClassifiers
       ->select(a|a.eAnnotations->exists(s|s.source = 'gmf.link'
       ))
11
12  inv: isDiagramDefined().oclIsUndefined()=false
13
14  inv: isDiagramDefined().oclIsUndefined()=false implies
       NodesAreDefined()
15
16  inv: isDiagramDefined().oclIsUndefined()=false implies
       ReferenceLinksAreDefined()
17
18
19 context EClass
20
21  def: linkSource():String= self.eAnnotations->select(s|s.
       source='gmf.link' and s.details->exists(k|k.key='source'
       and k.value.size()>0))
22
23  def: linkTarget():EAnnotation= self.eAnnotations-> select(s|
       s.source='gmf.link' and s.details->exists(k|k.key='taget'
        and k.value.size()>0))
24
25  def: ContainmentReferencesAreDefined():EBoolean= self.
       eAllStructuralFeatures->select(c | c.oclIsTypeOf(
       EReference))
26
27  inv IsValidSvgNode: self.eAnnotations->exists(s|s.source='
       gmf.node' and s.details-> exists(k|k.key='figure' and k.
```

```
        value='svg')) implies self.eAnnotations-> exists(s|s.
        source='gmf.node' and s.details->exists(k|k.key='svg.uri'
         and k.value.size()>0))
28
29      inv IsVlidPolygonNode: self.eAnnotations->exists(s|s.source=
        'gmf.node' and s.details->exists(k|k.key='figure' and k.
        value='polygon')) implies self.eAnnotations->exists(s,t|s
        .source='gmf.node' and t.source='gmf.node' and s.details
        ->exists(k|k.key='polygon.x' and k.value.size()>0)and t.
        details-> exists(k|k.key='polygon.y' and k.value.size()
        >0))
30
31      inv NodeLabelIsDefined: self.eAnnotations->exists(s|s.source
        ='gmf.node' and s.details->exists(k|k.key='label.
        placement' and k.value.size()>0)) implies self.
        eAnnotations->exists(s|s.source='gmf.node' and s.details
        -> exists(k|k.key='label' and k.value.size()>0))
32
33      inv LinkSourceIsDefined: self.eAnnotations->exists(s|s.
        source='gmf.link') implies self.eAnnotations->exists(s|s.
        source='gmf.link' and s.details-> exists(k|k.key='source'
         and k.value.size()>0))
34
35      inv LinkTargetIsDefined: self.eAnnotations->exists(s|s.
        source='gmf.link') implies self.eAnnotations->exists(s|s.
        source='gmf.link' and s.details-> exists(k|k.key='target'
         and k.value.size()>0))
36
37    context EAnnotation
38
39      inv isValidCompartment:self.source='gmf.compartment' implies
         self.eContainer().oclIsTypeOf(EReference)
40    endpackage
```

C EMG program for Eugenia Constraints

This section contains an EMG program that produces models that satisfy the Eugenia constraint discussed in Section 3.1.

```
1   operation EPackage create(){
2     self.name="ecore";
3   }
4
5   operation EClass create(){
6     var detail:Map = new Map;
7     if(randomBoolean()){
8       detail.put('rcp',Sequence{'true','false'}.uniRandom());
9     }
10    self.annotate("gmf.diagram",detail);
11    self.name=randomString();
12  }
13
14  $instances Sequence{1,nodeCount}
15  operation EClass create(){
16    self.name=randomString();
17    var detail:Map= new Map;
18    if(randomBoolean()){
19     var seq= Sequence{'rectangle', 'ellipse', 'rounded', 'svg'
          ,'polygon'};
20     var string:String =seq.uniRandom();
21     detail.put("figure",string);
22    if(string='polygon'){
23     detail.put("polygon.x",randomInteger(20)+"");
24     detail.put("polygon.y",randomInteger(20)+"");
25    }
26    else if(string='svg'){
27     detail.put("svg.uri",randomString());
28    }
29   }
30
31   //label
32    var sequence1= Sequence{'internal', 'external', 'none'};
33    var string:String = sequence1.uniRandom();
34    if(string<>'none'){
35     var label:String= "";
36     for(n in Sequence{1..randomInteger(1,4)}){
37      var r:EAttribute = new EAttribute;
38      r.name= randomString();
39      label=label+","+ r.name;
40      self.eStructuralFeatures.add(r);
41     }
42     detail.put("label",label.subString(1));
43     if(randomBoolean(0.75)){
44      detail.put("label.placement",string);
```

```
45              }
46            }
47            else{
48              detail.put("label.placement",string);
49            }
50            self.annotate("gmf.node",detail);
51          }
52
53          $instances Sequence{1,linkCount}
54          operation EClass create(){
55            self.name=randomString();
56            var detail:Map= new Map;
57            detail.put('source', randomString());
58            detail.put('target', randomString());
59            //label
60            var sequence1= Sequence{'internal', 'external', 'none'};
61            var string:String = sequence1.uniRandom();
62            if(string<>'none'){
63              var label:String= "";
64              for(n in Sequence{1..randomInteger(1,4)}){
65                var r:EAttribute = new EAttribute;
66                r.name= randomString();
67                label=label+","+ r.name;
68          self.eStructuralFeatures.add(r);
69              }
70              detail.put("label",label.subString(1));
71              if(randomBoolean(0.75)){
72                detail.put("label.placement",string);
73              }
74            }
75            else{
76              detail.put("label.placement",string);
77            }
78            self.annotate("gmf.link",detail);
79          }
80
81          $instances 5
82          operation EDataType create(){
83            self.name=randomString();
84          }
85
86          pattern package
87            pack:EPackage,pack2:EPackage,clas:EClass,clas2:EDataType
88            guard:pack.name="ecore" and pack2<>pack{
89              onmatch{
90                pack.eClassifiers.add(clas);
91                pack2.eClassifiers.add(clas2);
92                pack.eSubpackages.add(pack2);
93              }
```

```
94          }
95
96        pattern attribute
97          attr:EAttribute
98          guard: attr.etype.isUndefined(){
99           onmatch{
100            attr.eType=EDataType.all.uniRandom();
101           }
102         }
103
104       $number Sequence{1,EClass.all.select(t|t.isAnnotatedAs("gmf.
            node")).size()}
105       pattern node
106         root:EClass,node:EClass
107         guard : root.isAnnotatedAs("gmf.diagram") and node.
              isAnnotatedAs("gmf.node"){
108          onmatch{
109           var r= EReference.createInstance();
110           r.name=randomString();
111           r.eType=node;
112           root.eStructuralFeatures.add(r);
113          }
114         }
115
116       pattern link
117         root:EClass,link:EClass
118         guard : root.isAnnotatedAs("gmf.diagram") and
119         link.isAnnotatedAs("gmf.link"){
120          onmatch{
121           var r= EReference.createInstance();
122           r.name=randomString();
123           r.eType=link;
124           r.containment=true;
125           root.eStructuralFeatures.add(r);
126          }
127         }
128
129       pattern linkSource
130         class1:EClass
131         guard: class1.getAnnotationValue("gmf.link","source").
              isDefined(){
132          onmatch{
133           var r =EReference.createInstance();
134           r.name= class1.getAnnotationValue("gmf.link","source");
135           r.eType= EClass.all.excluding(class1).uniRandom();
136           class1.eStructuralFeatures.add(r);
137          }
138         }
139
```

```
140 pattern linkTarget
141   class1:EClass
142   guard:class1.getAnnotationValue("gmf.link","target").
          isDefined(){
143    onmatch{
144     var r =EReference.createInstance();
145     r.name= class1.getAnnotationValue("gmf.link","target");
146     r.eType=EClass.all.excluding(class1).uniRandom();
147     class1.eStructuralFeatures.add(r);
148    }
149   }
150
151 @probability 0.4
152 pattern nodeAnnotation
153   node:EAnnotation
154   guard:node.source="gmf.node" or node.source="gmf.link"{
155    onmatch{
156     var st:String= getKey(node.source);
157     var details:Map = new Map;
158     details.put(st,getValue(st));
159     node.addDetails(details);
160    }
161   }
162
163 @number 3
164 @probability 0.4
165 pattern isValidCompartment
166   ref: EReference{
167   onmatch{
168    var detail:Map= new Map;
169    if(randomBoolean()){
170     var st:String=Sequence{'collapsible','layout'}.uniRandom
          ();
171     detail.put(st,getValue(st));
172    }
173    ref.containment=true;
174    ref.annotate("gmf.compartment",detail);
175    }
176   }
177
178 // user-defined operations
179 operation EClass isAnnotatedAs(source:String){
180   return self.getEAnnotation(source).isDefined();
181 }
182 operation EClass getAnnotationValue(source:String,key:String)
      {
183   if(self.getEAnnotation(source).isDefined()){
184    var detail:EStringToStringMapEntry= self.getEAnnotation(
        source).details.selectOne(k|k.key=key);
```

```
185             if(detail.isDefined()){
186              var value:String = detail.value;
187              if(value.isDefined()){
188               return value;
189              }
190             }
191            }
192            return null;
193           }
194
195           operation EModelElement annotate(string:String){
196            self.annotate(string, new Map);
197           }
198
199           operation EModelElement annotate(string:String, details:Map){
200            var ann: new EAnnotation;
201            ann.source=string;
202            ann.addDetails(details);
203            self.eAnnotations.add(ann);
204           }
205
206           operation EAnnotation addDetails(details:Map){
207            for(d in details.keySet()){
208             var detail: new EStringToStringMapEntry;
209             detail.key=d;
210             detail.value=details.get(d);
211             self.details.add(detail);
212            }
213           }
214
215           operation getValue(string:String){
216            var st:String="";
217            if(string='border.style' or string= 'style'){
218             st= Sequence{'dot', 'dash', 'solid'}.uniRandom();
219            }
220            else if(string="layout"){
221             st=Sequence{'list', 'free'}.uniRandom();
222            }
223            else if(string='rcp' or string='label.icon' or string='
                  label.readOnly' or string='phantom' or string='resizable
                  ' or string='collapsible' or string='label.readOnly'){
224             st=Sequence{'true', 'false'}.uniRandom();
225            }
226            else if(string='border.color' or string='color'){
227             st= randomInteger(256)+","+randomInteger(256)+","+
                   randomInteger(256);
228            }
229            else if(string='size'){
230             st= randomInteger(20)+","+randomInteger(20);
```

```
231           }
232           else if(string = 'border.width' or string='margin'){
233            st=randomInteger(20)+"";
234           }
235           else if (string="source.decoration" or string="target.
                 decoration"){
236            st = Sequence{'none', 'arrow', 'rhomb', 'filledrhomb','
                 square', 'filledsquare', 'closedarrow', '
                 filledclosedarrow'}.uniRandom();
237           }
238           return st;
239         }
240
241         operation getKey(string:String){
242           var st:String="";
243           if(string="gmf.node"){
244            st = Sequence{'border.width','margin','size','border.color
                 ','color', 'border.style','label.icon','label.readOnly'
                 ,'phantom','resizable'}.uniRandom();
245           }
246           else if(string="gmf.link"){
247            st = Sequence{'source.decoration','target.decoration','
                 color', 'style'}.uniRandom();
248           }
249           return st;
250         }
```

D Figures for EMG Tools

This section contains graphical diagrams of available tools in EMG as described in Section 5.7. Figure D.1 is a sample EPL editor while Figure D.2 is a sample Epsilon console. Figures D.3, D.4, D.5 and D.6 are sample tabs of the runtime configuration tool described in Section 5.7.3.

Figure D.1: EPL Editor

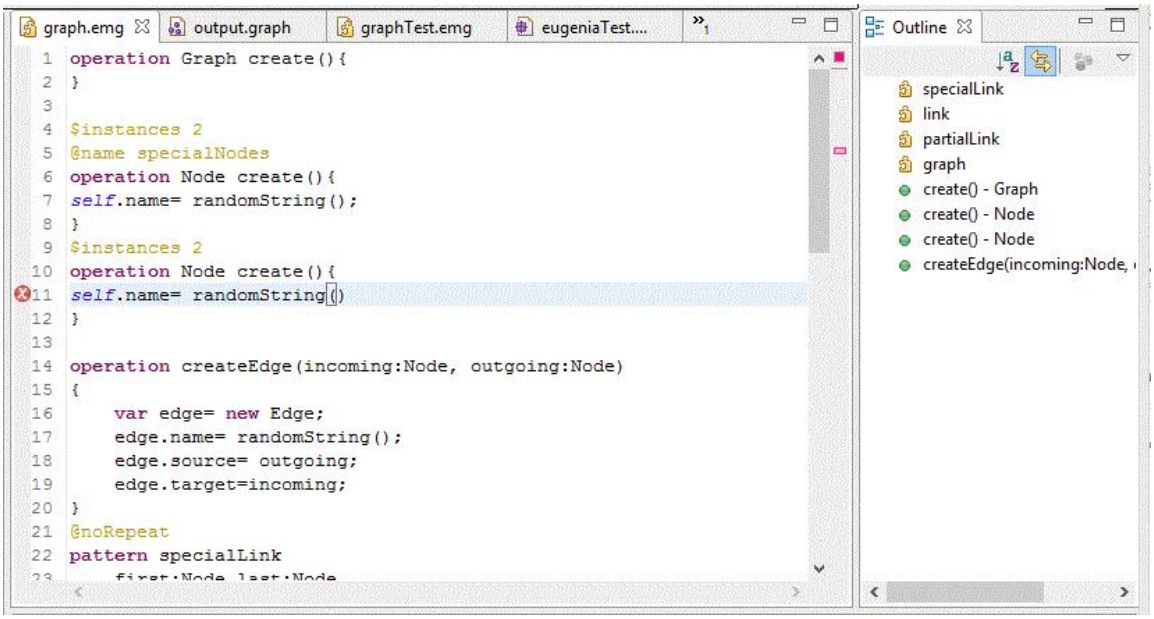


Figure D.2: Epsilon Console

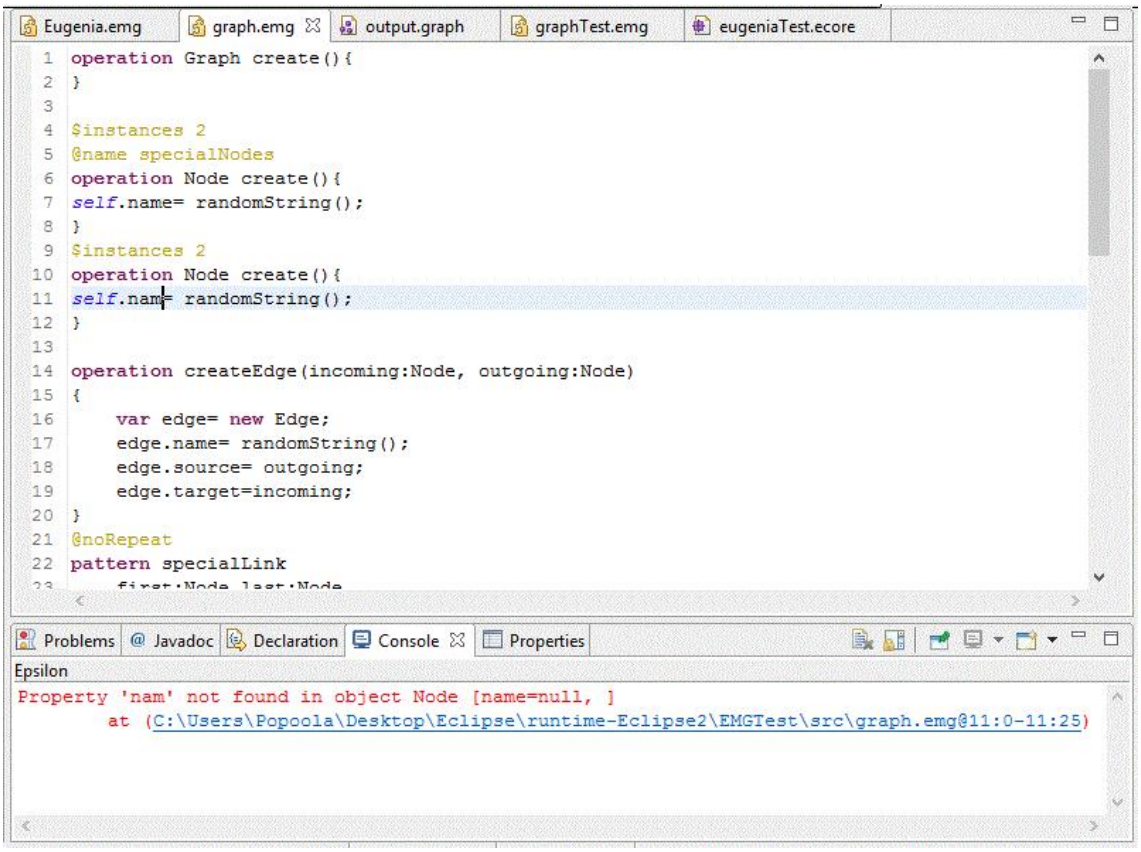


Figure D.3: Run Configuration Interface: Source Tab

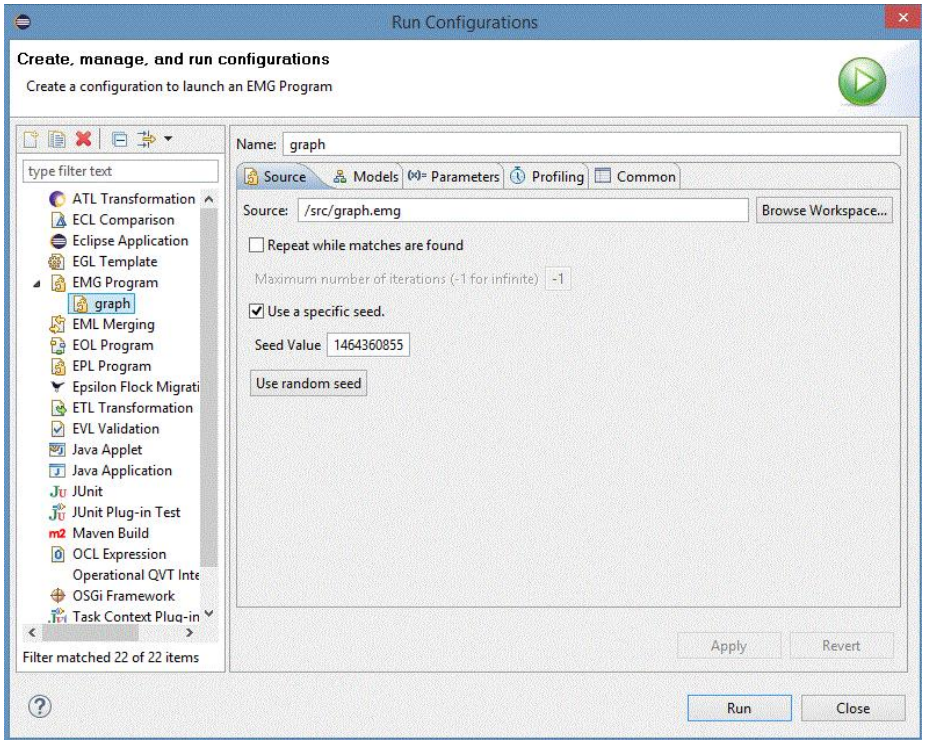


Figure D.4: Run Configuration Interface: Models Tab

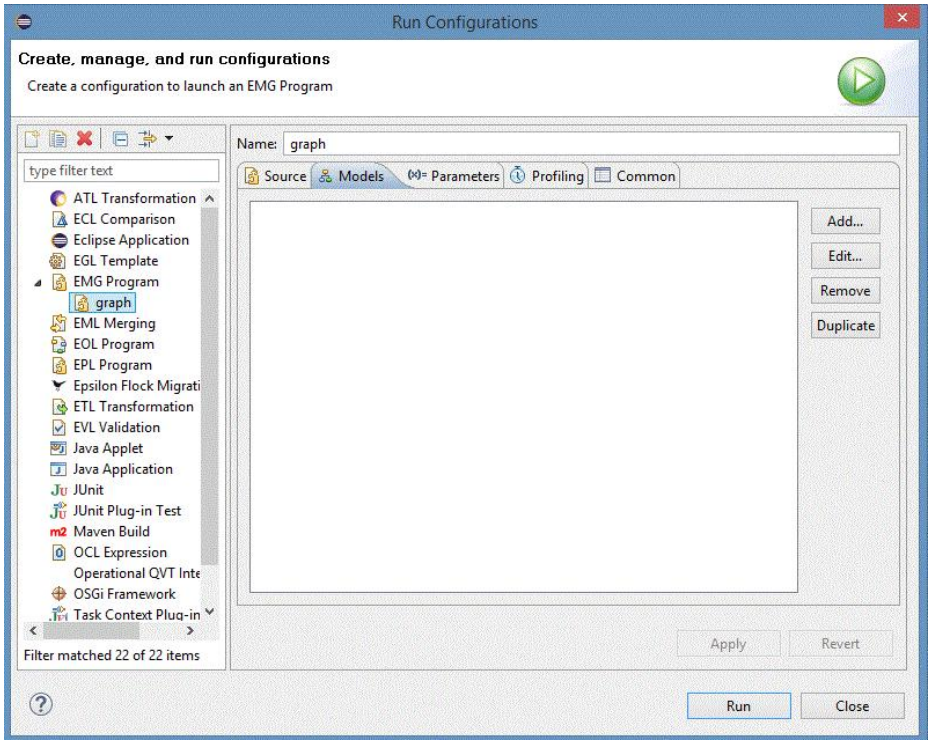


Figure D.5: New Model Configuration Interface

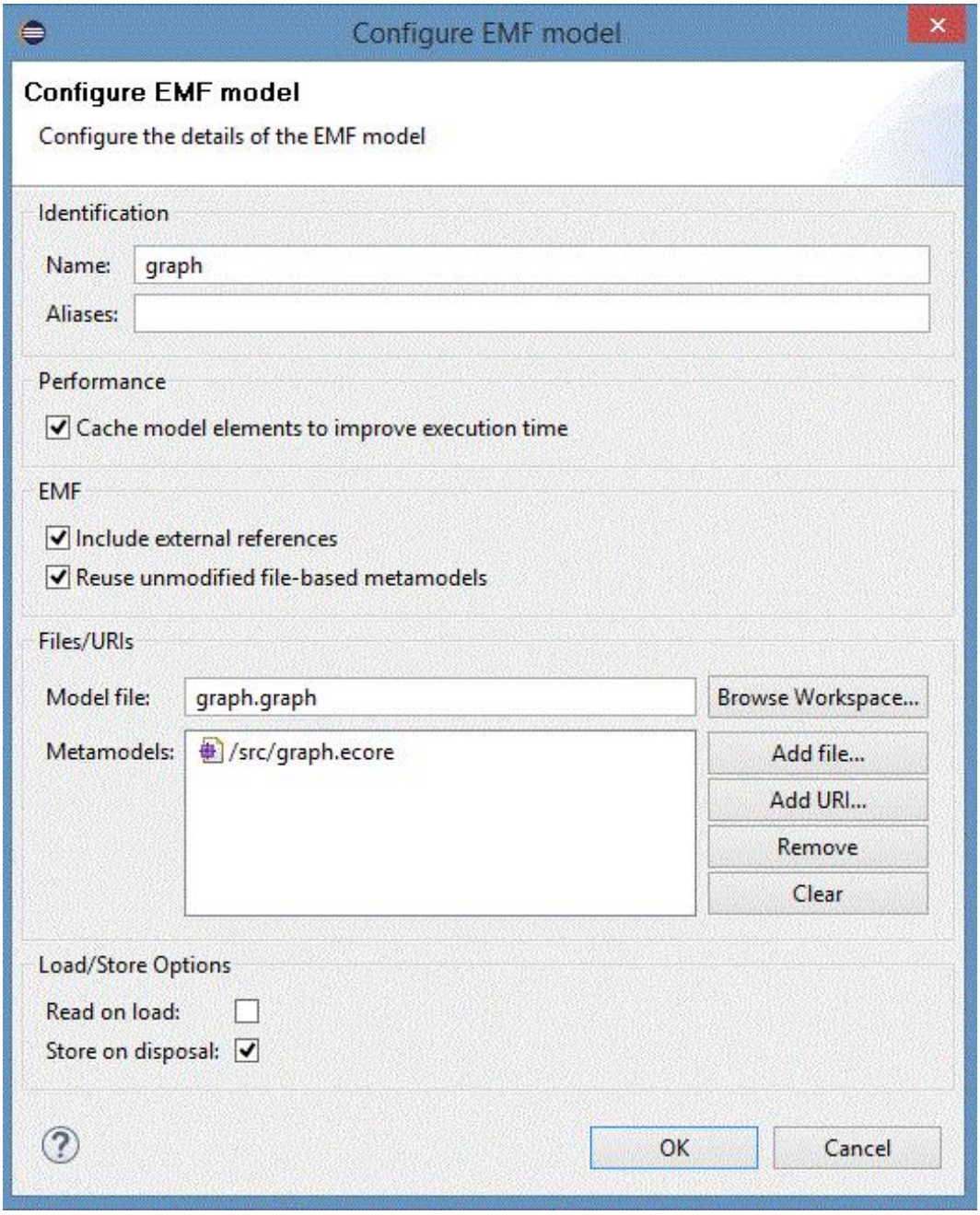
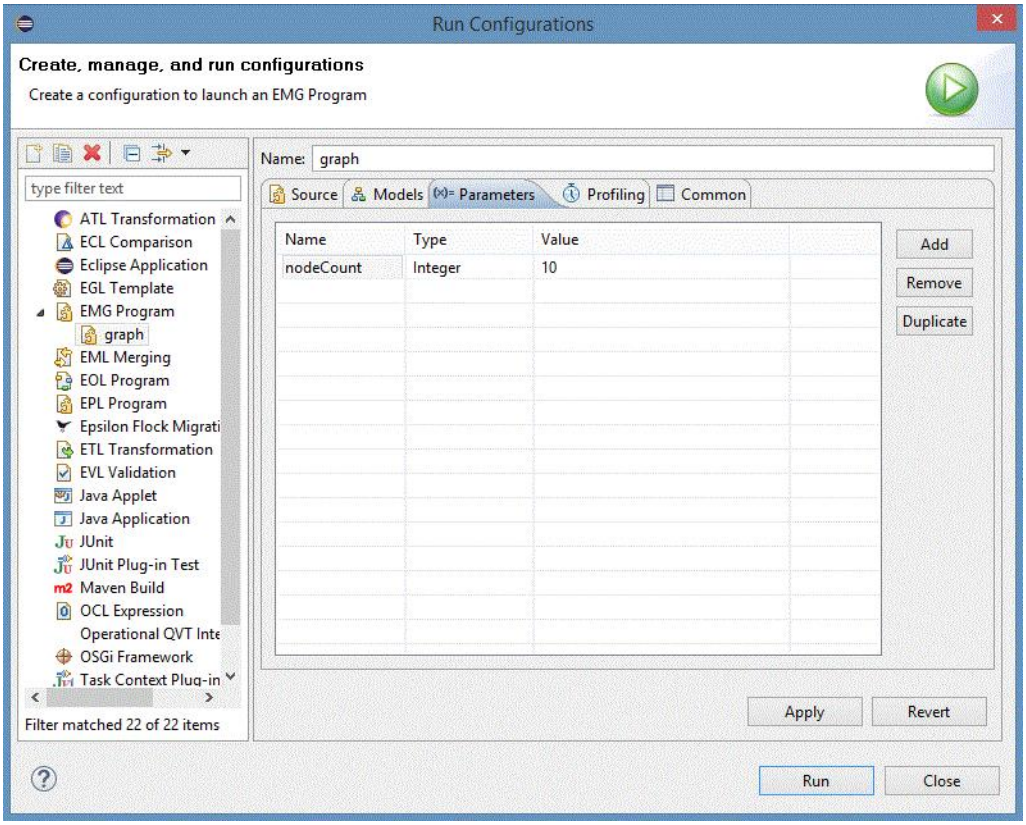


Figure D.6: Run Configuration Interface: parameters Tab



E EMG and EOL Code for Generating Railway Models

This section contains EMG and EOL code for generating railway models described in Section 6.3.1.

Listing 1: A Sample EMG Railway Generator

```
operation RailwayContainer create(){
}

$instances semaphore
operation Semaphore create(){
  self.signal= Sequence{Signal#FAILURE,Signal#STOP}.randomD()
      ;
}

$instances Sequence{1..route}
  operation Route create(){
}

$instances position
  operation Sensor create(){
}

$instances Sequence{1..swit}
operation Switch create(){
  self.Position= Sequence{Position#FAILURE,Position#LEFT,
      Position#RIGHT,Position#STRAIGHT}.randomD();
}

$instances segment
operation Segment create(){
  self.length= randomInteger();
}

$instances position*2
operation SwitchPosition create(){
  self.currentPosition= Sequence{Position#FAILURE,Position#
      LEFT,Position#RIGHT,Position#STRAIGHT}.randomD();
}

pattern SemaphoreNeighbour
  track1:TrackElement,track2:TrackElement
  from: TrackElement.all.select(t|t.connectsTo.size()>0 and t
      .sensor.isDefined())
  guard: track1.connectsTo.includes(track2),route1:Route,
      route2:Route
  guard: route1<>route2 and route1.definedBy.includes(track1.
      sensor)and route2.definedBy.includes(track2.sensor){
   onmatch{
```

```
38          var s:Semaphore= new Semaphore;
39          s.signal= Sequence{Signal#FAILURE,Signal#STOP}.randomD();
40          route1.exit=s;
41          route2.entry=s;
42         }
43        }
44
45      @probability 0.8
46      pattern SwitchSet
47        swP:SwitchPosition,sw:Switch,route:Route
48        guard: route.follows.includes(swP) and sw.positions.
              includes(swP) and swP.position=sw.currentPosition{
49         onmatch{
50          route.entry.signal=Signal#GO;
51         }
52        }
53
54      pattern RailwayContainer
55        railway:RailwayContainer,route:Route,sem:Semaphore
56        guard:railway.routes.excludes(route) and railway.semaphores
              .excludes(sem){
57         onmatch{
58          railway.routes.add(route);
59          railway.semaphores.add(sem);
60         }
61        }
62
63      @noRepeat
64      pattern Invalids
65        rail:RailwayContainer,route:Route,sw:Switch,swPos:
              SwitchPosition,seg:Segment
66        guard: (route.entry.isDefined() or route.exit.isDefined()
              or
67        route.definedBy.size()>1) and (swPos.route.isDefined or
68        swPos.switch.isDefined) and sw.sensor.isDefined and seg.
              length>0 and
69        rail.invalids.excludeAll(Sequence{route,sw,swPos,seg}){
70         onmatch{
71          var railway= RailwayContainer.all.randomD();
72          railway.invalids.addAll(Sequence{route,sw,swPos,seg});
73         }
74        }
```

Listing 2: A Sample EOL Railway Generator

```
1       var railway = new RailwayContainer;
2       for (i in 1.to(randomNumber(1,node))){
3         var route= new Route;
4       }
5
```

```
6         for (i in 1.to(randomNumber(1,node))){
7           var sensor= new Sensor;
8         }
9
10        for (i in 1.to(randomNumber(1,node))){
11          var swPosition= new SwitchPosition;
12          sw.currentPosition= Sequence{Position#FAILURE, Position#LEFT
                ,Position#RIGHT, Position#STRAIGHT}.randomD();
13        }
14
15        for (i in 1.to(randomNumber(1,node))){
16          var segment= new Segment;
17          segment.length= randomInteger(500000);
18        }
19
20        for (i in 1.to(randomNumber(1,node))){
21          var sw= new SwitchPosition;
22          sw.currentPosition= Sequence{Position#FAILURE, Position#LEFT
                ,Position#RIGHT, Position#STRAIGHT}.randomD();
23        }
24
25        // semaphore neighbour constraint
26        for (track1 in TrackElement.all()){
27         for (track2 in TrackElement.all()){
28          if(track1.connectsTo.includes(track2) and track2.connectTo.
                size()>0){
29           for(route1 in Route.all){
30            for (route2 in Route.all){
31             if(route1<>route2 and route1.definedBy.includes(track1.
                    sensor) and route2.definedBy.includes(track2.sensor))
                    {
32              var s:Semaphore= new Semaphore;
33              s.signal= Sequence{Signal#FAILURE, Signal#STOP, Signal#GO
                    }.randomD();
34              route1.exit=s;
35              route2.entry=s;
36             }
37            }
38           }
39          }
40         }
41        }
42
43        //switchset
44        for(swp in SwitchPosition.all){
45         for(sw in Switch.all){
46          for(route in Route.all){
47           if(route.follows.includes(swP) and sw.positions.includes(
                swP) and swP.position=sw.currentPosition){
```

```
48              if(randomInteger(10)<8){
49               route.entry.signal=Signal#GO;
50              }
51             }
52            }
53           }
54          }
55
56          //railwayContainer
57          for (route in Route.all){
58           if(railway.routes.excludes(route)){
59            railway.routes.add(route);
60           }
61          }
62          for (sem in Semaphore.all){
63           if(railway.semaphores.excludes(sem)){
64            railway.semaphores.add(sem);
65           }
66          }
67
68          //invalids
69          for(route in Route.all){
70           if(route.entry.isUndefined() or route.exit.isUndefined() or
                route.definedBy.size()<2){
71            if(railway.invalids.excludes(route)){
72             railway.invalids.add(route);
73            }
74           }
75          }
76          for(sw in Switch.all ){
77           if(sw.sensor.isUndefined){
78            if(railway.invalids.exclude(sw)){
79             railway.invalids.add(sw);
80            }
81           }
82          }
83          for(swPos in SwitchPosition.all ){
84           if(swPos.'switch'.isUndefined){
85            if(railway.invalids.exclude(swPos)){
86             railway.invalids.add(swPos);
87            }
88           }
89          }
90          for(seg in Segment.all ){
91           if(seg.length.size<0){
92            if(railway.invalids.exclude(seg)){
93             railway.invalids.add(seg);
94            }
95           }
```

```
96          }
97
98          operation randomNumber(low:Integer,high:Integer){
99           var num:Sequence=Sequence{low..high};
100          return num.random();
101         }
102         operation randomString(){
103          var letter:String="abcdefghijklmnopqrstuvwxyz";
104          var st:String="";
105          for (i in 5.to(20)){
106           st = st.concat(letter.charAt(randomNumber(0,25))+"");
107          }
108          return st;
109         }
```

Bibliography

[1] Jochen Ludewig. Models in software engineering - an introduction. *Software and Systems Modeling*, 2:5–14, 2002.

[2] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[3] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[4] Object Management Group. UML official website. `http://www.uml.org`.

[5] Jordi Cabot and Martin Gogolla. Object constraint language (ocl): A definitive guide. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *SFM*, volume 7320 of *Lecture Notes in Computer Science*, pages 58–90. Springer, 2012.

[6] Mark Richters and Martin Gogolla. *OCL: Syntax, Semantics, and Tools*, pages 42–68. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[7] Kolovos Dimitris, Rose Louis, Garcaa Domanguez Antonio, and Paige Richard. The Epsilon book. http://eclipse.org/epsilon/doc/book.

[8] John Daniels. Modeling with a sense of purpose. *IEEE Software*, 19(1):8–10, 2002.

[9] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19, 2003.

[10] Meta-Object Facility. `http://www.omg.org/mof/`.

[11] Jack Greenfield and Keith Short. Software factories: Assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 16–27, New York, NY, USA, 2003. ACM.

[12] Martin Matula. NetBeans metadata repository. `http://netbeans-org.1045718.n5.nabble.com/Netbeans-MDR-f3045597.html`, 2003.

[13] Steven Kelly, Kalle Lyytinen, and Matti Rossi. *MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment*, pages 1–21. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[14] Eclipse Modelling Framework. `http://www.eclipse.org/modeling/emf/`.

[15] Eclipse Graphical Modeling Framework , official website. http://www.eclipse.org/gmf-tooling.

[16] ATL - a model transformation technology. http://www.eclipse.org/atl.

[17] Acceleo model to text language. https://eclipse.org/acceleo.

[18] Xtend programming language. http://www.eclipse.org/xtend/.

[19] VIATRA: an event-driven and reactive model transformation platform. http://www.eclipse.org/viatra.

[20] Dimitrios Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Object Language (EOL). volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2006.

[21] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Transformation Language. In *Theory and Practice of Model Transformations: First International Conference, ICMT 2008, Zurich, Switzerland*, pages 46–60. Springer Berlin Heidelberg, 2008.

[22] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. On the evolution of OCL for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis*, volume 5115 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2009.

[23] Dimitrios S. Kolovos, Richard F. Paige, Fiona Polack, and Louis M. Rose. Update transformations in the small with the epsilon wizard language. *Journal of Object Technology*, 6(9):53–69, 2007.

[24] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. The Epsilon Generation Language. In *ECMDA-FA*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

[25] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model comparison: A foundation for model composition and model transformation testing. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, GaMMa '06, pages 13–20, New York, NY, USA, 2006. ACM.

[26] Dimitrios Kolovos, Richard F. Paige, and Fiona Polack. Merging models with the Epsilon Merging Language (EML). pages 215–229. 2006.

[27] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, Fiona A. C. Polack, and Simon M. Poulding. Epsilon Flock: a model migration language. *Software and System Modeling*, 13(2):735–755, 2014.

[28] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley and Sons, 2004.

[29] Whittaker J.A. What is software testing? and why is it so hard? *IEEE Software*, 17(1):70–79, Jan 2000.

[30] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[31] Edward E Ogheneovo. Software dysfunction: Why do software fail? *Journal of Computer and Communications*, 2:25–35, 2014.

[32] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge,UK, 2008.

[33] Boris Beizer. *Black-box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley and Sons, Inc., New York, NY, USA, 1995.

[34] Farmeena Khan Mohd Ehmer Khan. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications(IJACSA)*, 3(6), 2012.

[35] Srinivas Nidhra and Jagruthi Dondeti. Blackbox and whitebox testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[36] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, sixth edition, 2005.

[37] Tafline Murnane and Karl Reed. On the effectiveness of mutation analysis as a black box testing technique. In *Australian Software Engineering Conference*, pages 12–20. IEEE Computer Society, 2001.

[38] Carlos A. Gonzalez and Jordi Cabot. ATLTest: a white-box test generation approach for ATL transformations. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 449–464, Berlin, Heidelberg, 2012. Springer-Verlag.

[39] Robert L Probert. Grey-box (design based) testing techniques. In *Proceedings of 15th Hawaii Int. Conf. on System Sciences*, pages 94–102, 1982.

[40] Kiumi Akingbehin. Towards destructive software testing. In *Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering,Software Architecture and Reuse*, ICIS-COMSAR '06, pages 374–377. IEEE Computer Society, 2006.

[41] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[42] Hareton KN Leung and Lee White. Insights into regression testing [software testing]. In *Proceedings of 1989 Conference on Software Maintenance*, pages 60–69. IEEE, 1989.

[43] Elaine J Weyuker and Filippos I Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE transactions on software engineering*, 26(12):1147, 2000.

[44] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[45] Zary Segall, D Vrsalovic, D Siewiorek, D Ysskin, J Kownacki, J Barton, R Dancey, A Robinson, and T Lin. Fiat-fault injection based automated testing environment. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, page 394. IEEE, 1995.

[46] Henrique Madeira, Diamantino Costa, and Marco Vieira. On the emulation of software faults by software fault injection. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 417–426. IEEE, 2000.

[47] James M Bieman, Daniel Dreilinger, and Lijun Lin. Using fault injection to increase software test coverage. In *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, pages 166–174. IEEE, 1996.

[48] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments?[software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411. IEEE, 2005.

[49] Martin R Woodward. Mutation testing- its origin and evolution. *Information and Software Technology*, 35(3):163–169, 1993.

[50] J.M. Jezequel, D. Deveaux, and Y. Le Traon. Reliable objects: Lightweight testing for oo languages. *IEEE Software*, 18(4):76–83, 2001.

[51] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6):139–143, June 2010.

[52] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *17th International Symposium on Software Reliability Engineering ISSRE '06*, pages 85–94, 2006.

[53] Muhammad Uzair Khan Atif Aftab Jilani, Muhammad Zohaib Iqbal. A search based test data generation approach for model transformations. In *7th International Conference on Model Transformations*, 2014.

[54] Jiang W Wang W., Kessentini M. Test cases generation for model transformations from structural information. In *17th European Conference on Software Maintenance and Reengineering, Genova, Italy (2013)*, Genova, Italy, 2013.

[55] Adel Ferdjoukh, Anne-Elisabeth Baert, Eric Bourreau, Annie Chateau, Remi Coletta, and Clementine Nebut. Instantiation of meta-models constrained with OCL

- a CSP approach. In *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*, pages 213–222, Feb 2015.

[56] Franck Fleurey, Baudry Benoit, Muller Pierre-Alain, and Y. Le Traon. Qualifying input test data for model transformations. *Software & Systems Modeling*, 8(2):185–203, Apr 2009.

[57] George Devaraj Heimdahl, Mats P.E. Test-suite reduction for model based tests: Effects on test quality and implications fro. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE 04, pages 176–185. IEEE Computer Society, 2014.

[58] Gehan M. K. Selim, James R. Cordy, and Juergen Dingel. Model transformation testing: The state of the art. In *Proceedings of the First Workshop on the Analysis of Model Transformations*, AMT '12, pages 21–26, New York, NY, USA, 2012. ACM.

[59] Yuehua Lin, Jin Zhang, and Jeff Gray. Mode comparism: A key challenge for model transformation testing and version control in model driven software development. In *Control in Model Driven Software Development. OOPSLA/GPCE: Best practices for Model-Driven Software Development*, pages 219–236. Springer, 2014.

[60] M Gogolla and A Vallecillo. Tractable model transformation testing. In *ECMFA*, 2011.

[61] Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michele Soria. Uniform random generation of huge metamodel instances. In *Model Driven Architecture - Foundations and Applications*, number 5562 in Lecture Notes in Computer Science, pages 130–145. Springer Berlin Heidelberg, 2009.

[62] S. Ali, M.Z. Iqbal, A. Arcuri, and L. Briand. A search-based OCL constraint solver for model-based test data generation. In *11th International Conference on Quality Software (QSIC)*, pages 41–50, 2011.

[63] Camillo Fiorentini, Alberto Momigliano, Mario Ornaghi, and Iman Poernomo. A constructive approach to testing model transformations. In *Theory and Practice of Model Transformations*, number 6142 in Lecture Notes in Computer Science, pages 77–92. Springer Berlin Heidelberg, 2010.

[64] C.A. Gonzalez, F. Buttner, R. Clariso, and J. Cabot. EMFtoCSP: A tool for the lightweight verification of EMF models. In *Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012 Formal Methods in*, pages 44–50, 2012.

[65] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic model generation strategies for model transformation testing. In Richard F. Paige, editor, *Theory and Practice of Model Transformations*, number 5563 in Lecture Notes in Computer Science, pages 148–164. Springer Berlin Heidelberg, 2009.

[66] Williams James and Poulding. Simon. Generating models using metaheuristic search. In *Proceedings of the Fourth York Doctoral Symposium on Computing, York*, pages 53–60, 2014.

[67] Markus Scheidgen. Generation of large random models for benchmarking. In *Proceedings of the 3rd Workshop on Scalable Model Driven Engineering*, pages 1–10, L'Aquila, Italy, 2015.

[68] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M Kuster. Analysis of model transformations via Alloy. In *4th MoDeVVa workshop, Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007.

[69] A. Ferdjoukh, A.-E. Baert, A. Chateau, R. Coletta, and C. Nebut. A CSP approach for metamodel instantiation. In *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, pages 1044–1051, 2013.

[70] Karsten Ehrig, Jochen Malte Kuster, and Gabriele Taentzer. Generating instance models from meta models. *Software & Systems Modeling*, 8(4):479–500, 2008.

[71] He Xiao, Zhang Tian, Ma Zhiyi, and Shao Weizhong. Randomized model generation for performance testing of model transformations. In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 11–20, 2014.

[72] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`.

[73] Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing*, 13(4-5):577–625, July 2004.

[74] Dimitrios S. Kolovos, Louis M. Rose, Saad Bin Abid, Richard F. Paige, Fiona A. C. Polack, and Goetz Botterweck. Taming EMF and GMF using model transformation. In Dorina C. Petriu, Nicolas Rouquette, and Oystein Haugen, editors, *Model Driven Engineering Languages and Systems*, number 6394 in Lecture Notes in Computer Science, pages 211–225. Springer Berlin Heidelberg, 2010.

[75] Sylvain Merchez, Christophe Lecoutre, and Frederic Boussemart. Abscon: A prototype to solve CSPs with abstraction. In *Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 730–744. Springer Berlin Heidelberg, 2001.

[76] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[77] Kermeta: A language for metamodel engineering. `http://www.kermeta.org/`.

[78] Hao Wu, R. Monahan, and J.F. Power. Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In *2013 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 175–182, 2013.

[79] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[80] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[81] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[82] Jurgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol: An interpolating SMT solver. In *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, pages 248–254, 2012.

[83] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, and Fiona A. C. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '09, pages 162–171. IEEE Computer Society, 2009.

[84] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006.

[85] Raymond P.L. Buse and Westley R. Weimer. A metric for software readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 121–130, New York, NY, USA, 2008. ACM.

[86] Ahmad Salim Al-Sibahi. On the computational expressiveness of model transformation languages. *ITU Technical Report Series*, 2015.

[87] Richard F. Paige, Louis M. Rose, Xiaocheng Ge, Dimitrios S. Kolovos, and Phillip J. Brooke. *FPTC: Automated Safety Analysis for Domain-Specific Languages*, pages 229–242. Springer Berlin Heidelberg, 2009.

[88] Model-based generation of tests for dependable embedded systems,7th EU framework programme. `http://http://mogentes.eu/`, 2011.

[89] Ecore metamodel. `http://download.eclipse.org/modeling/emf/emf/javadoc?org/eclipse/emf/ecore/package-summary.html`.