A DSL for defining instance templates for the ASMIG system

Andrii Kovalov

Dissertation 2014

Erasmus Mundus MSc in Dependable Software Systems

Department of Computer Science
National University of Ireland, Maynooth
Co. Kildare, Ireland

A dissertation submitted in partial fulfilment
of the requirements for the
Erasmus Mundus MSc Dependable Software Systems

Head of Department :	Dr Adam Winstanley
Supervisor :	Dr James Power
Date:	June 2014

Abstract

The area of our work is test data generation via automatic instantiation of software models. *Model instantiation* or *model finding* is a process of finding instances of software models. For example, if a model is represented as a UML class diagram, the instances of this model are UML object diagrams. Model instantiation has several applications: finding solutions to problems expressed as models, model testing and test data generation. There are systems that automatically generate model instances, one of them is ASMIG (A Small Metamodel Instance Generator). This system is focused on a 'problem solving' use case. The motivation of our work is to adapt ASMIG system for use as a test data generator and make the instance generation process more transparent for the user. In order to achieve this we provided a way for the user to interact with ASMIG internal data structure, the *instance template graph* via a specially designed graph definition domain-specific language. As a result, the user is able to configure the instance template in order to get plausible instances, which can be then used as test data. Although model finding is only suitable for obtaining test inputs, but not the expected test outputs, it can be applied effectively for smoke testing of systems that process complex hierarchic data structures such as programming language parsers.

Declaration

I declare that this report, which I now submit for assessment on the program of study leading to the award of Master of Science in Dependable Software Systems, is entirely my own work and has not been taken from the work of the others except for the cited fragments.

Andrii Kovalov

Acknowledgement

I want to thank my supervisor Dr James Power whose room I entered sometimes sad, but always exited inspired and full of enthusiasm.

Contents

Chapter 1

Introduction

This report is mainly about software models and the automated instantiation of these models. If we have, for example, a model of some data structure or some language, which abstracts over their common properties, it is possible to automatically generate examples of the data stored in the structure or the expressions in the language respectively. This generation is called *model instantiation* or *model finding*.

The main use cases of automated model instantiation are:

1. Finding solutions to problems that can be expressed as the model finding problems.

2. Generating test data.

3. Testing the correctness of the model itself.

There are several systems that automatically perform this instance generation on being supplied with the model expressed in some modelling language. These tools usually focus on performing one of these three use cases. We will be talking more closely about the systems Alloy, USE and ASMIG.

The ASMIG system is a system that is being developed in the Computer Science department of National University of Ireland, Maynooth and it is primarily focused on the first use case - solving problems represented in the form of model finding task.

The 'big picture' of ASMIG work flow is shown in Figure 1.1.

The main motivation for our work is to adapt the ASMIG system for use as a test data generator (use case 2 in the list above). The current version of ASMIG is unsuitable for this purpose because it lacks the precise control over the instance generation process that we would like to have in order to use the resulting instances as test data.

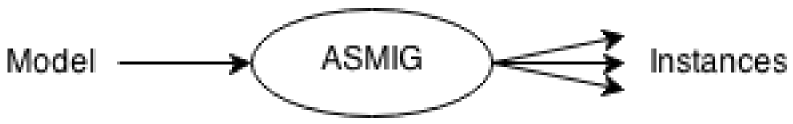


Figure 1.1: Big picture of ASMIG work flow. ASMIG generates instances of a model

The second motivation is to make instance generation in ASMIG more transparent for the user. Figure 1.1 represents the user interaction with the system quite fully - the user gives the input (the model) and gets the output (the instances). The current system itself is a black box and there is no way to influence or simply track the generation process.

Our way to achieve these two goals of making the system more transparent and making it suitable for test data generation is illustrated in Figure 1.2.

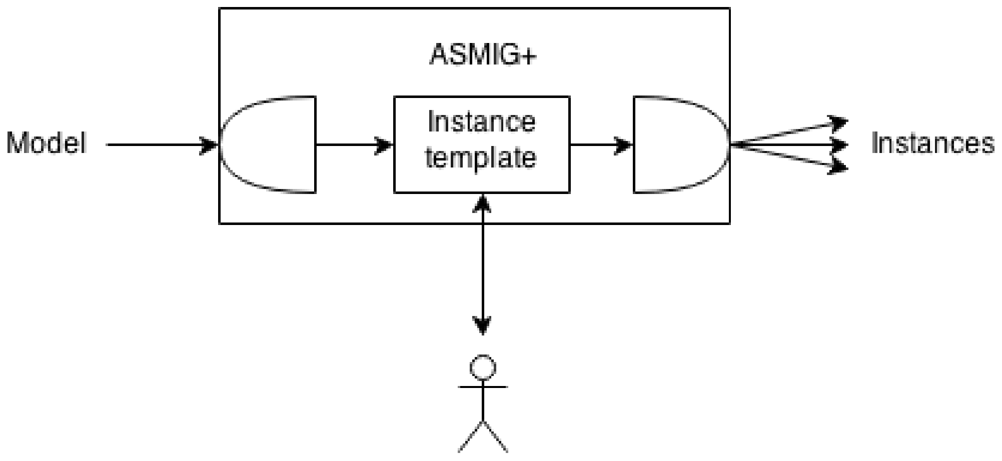


Figure 1.2: ASMIG+: the modified ASMIG where the user can interact with the instance template inside ASMIG

Here we need to explain that the ASMIG system builds an instance template for a model. This template is represented as a graph. The generated instances produced by the ASMIG system are represented as the subgraphs of this template graph.

In our extended system built on the top of ASMIG (we called it ASMIG+) the user has direct access to this instance template graph and can modify it.

By editing the instance template the user can get the instances with certain predefined properties. This is necessary for the test data generation scenario as the real test data should not be random and should contain some plausible values.

To provide a way of editing the instance template graph we designed a special domain-specific language. In our ASMIG+ system the user can now export the instance template graph in the form of a text script in our language, then edit it and then send it back as an input to the ASMIG system. So our work in this thesis mainly deals with the design of this language and the integration of it into the existing ASMIG system.

The report has the following structure: Chapter 2 gives an overview of metamodeling and model instantiation, compares the existing model instantiating systems, discusses the area of Domain Specifig Languages and specifically the languages related to graphs. Chapter 3 talks about the engineering of the ASMIG+ system and the integration into the existing ASMIG system. In Chapter 4 we evaluate our work by performing system testing and demonstrating a use case of test data generation with the ASMIG+ system. Chapter 5 concludes the report and outlines the potential future work.

Chapter 2

Related Work

This chapter covers material which is related to the current work. First of all we will give an overview of the model instantiation problem, then discuss and compare which existing systems perform this task - Alloy, USE and ASMIG. The ASMIG system will be examined closely as the scope of the project is to extend this system with new functionality. Finally, we will discuss the area of Domain-Specific Language engineering and have a look at the graph definition languages, since a considerable part of our work is to design a language for defining instance templates for the ASMIG system.

2.1 Metamodels and model instantiation

First of all we would like to define the terminology that we will be using. Our work primarily deals with models and model instantiation. Sometimes a term metamodel is used. While with the word *model* everything is clear, the word *metamodel* tends to be somewhat confusing. The concise definition is given in [1]: ‘A metamodel is a model used to specify a language’. Here are some other definitions one might encounter in literature: ‘explicit specification of an abstraction expressed in a specific language’[2], ‘model of the conceptual foundation of a modeling language’[3], ‘a specification model for a class of systems where each system in the class is itself a valid model expressed in a certain modeling language’[4]. Further discussion on the terminology can be found in [5]. In the current work the word *metamodel* will be used to show that the particular model describes a language (not necessarily a modelling language). For example, the Java metamodel is a model that contains elements of the Java language. A term *model* when used in the text means any model including a metamodel.

Metamodels serve two purposes. First of all, they can be used to verify models. In other words, to check if the programme written in a particular language is valid. The example of this use case is error highlighting in an IDE. The code in the editor is matched against the language metamodel and if it is not fully compliant, there is an error. A second purpose of a metamodel is instance generation (also referred to as model instantiation or model finding; we will be using all these expressions interchangeably).

Instance generation is easy to explain using an example. Let's assume we have a UML class diagram of some system, probably with some constraints specified in Object Constraint Language (OCL) [6] . It is a model of a system. The instance of this model will then be an object diagram containing the objects with types from the class diagram, related appropriately, and where all the OCL constraints hold. Similarly, if we have a metamodel which corresponds to some language, an instance of this metamodel would be a valid expression in this language. For example, an instance of the UML metamodel will be some UML model. The instance of the Java metamodel are some Java objects. Some models have infinitely many possible instances, other have a finite number of instances. It is also possible that a model doesn't have any instances. This means that there is some contradiction in it.

Model instantiation has two motives: they are, problem solving via finding a model instance and test data generation. To solve problems using model instantiation the problem needs to be expressed as a model where the instances of this model are problem solutions. For example, in [7] the authors address the problem of network configuration. They build a model of a desired network with all the requirements and constraints and then use the Alloy tool to find an instance of this model which represents one possible configuration of network nodes. In this case the problem of configuring a network is reduced to a problem of model finding. Whenever such reduction is possible one can use model finding tools to solve the problem. The second usage of model instantiation is the generation of test data. To put it clearly we will again use an example. Assume we have a bank system that can process different types of transactions. To test this system we need some test transactions. Of course we start with careful test design, and write test cases that cover all the specification. However, we are limited when we create test transactions manually, whereas we could automatically generate millions of different test transactions. In order to do this we create a model of a transaction. Having a tool which performs model instantiation we can generate as many different instances of this model (transactions) as we need.

The instantiation of metamodels also makes sense. If we are developing a compiler for some programming language we would like to test it using programs written in this language. As defined above, programs are instances of this language metamodel, so if we have a metamodel and instance generator we can generate test programs for our compiler.

The aspect we would like to emphasize here is that in order to use such a system we need to have some control over the instances that will be generated, as without any control we are likely to get a set of instances that don't differ much or which have parameters that are not plausible.

As we can see, instance generation is a practical problem and there are tools that perform this task. In this overview we will look more closely to three such systems - Alloy, USE and ASMIG. Most of the tools reduce the model finding problem to the SAT problem and then use a SAT solver. However, we need to mention that this is not the only possible way. An alternative approach is illustrated in [8] and uses graph grammars with a set of production rules.

2.2 Model Instantiation Systems

2.2.1 Alloy

The Alloy system[1] is a project of Software Design Group at Massachusetts Institute of Technology. Alloy allows us to create models using our own declarative specification language. The language has rich support for specifying constraints on model, similar to OCL. Instance generation in Alloy is automatic and is done within a bounded scope. Also one of applications of Alloy is checking model properties by finding counter-examples (in the bounded search space).

The instance generation mechanism in Alloy works by translating the model and all the constraints to a boolean formula and using a SAT solver to find out whether it is satisfiable or not. If it is satisfiable the SAT solver gives the values to the boolean variables in the formula which represent an instance of a model. All the details on the language and the logic behind it, as well as a lot of examples, can be found in a book by project director Daniel Jackson [9].

A question that should interest us in relation to our current work is the amount of control the user has over the instances to be generated. First of all, by using predicates, the user can bound the number of objects of every signature[2] as well as the arity of the relations. However, Alloy lacks support of primitive types. The only primitive type supported is integer, and the only way to control the value of an integer is to set the maximum bit-width.

[1]`http://alloy.mit.edu/alloy/`

[2]In Alloy terminology a signature is a classifier (similar to a class in UML).

For instance, if the bit-width is set to 6, the integer will get a value between -31 and 31. Other primitive types such as booleans, floats and strings have no native support.

The absence of support for primitive types makes Alloy unsuitable for test data generation. This should not be regarded as a flaw of the system as its main goal is model exploration and analysis, and not instance generation; this is explicitly discussed in [9, pp. 135-137].

2.2.2 USE

Another system we would like to draw our attention to and have a slightly closer look at is USE: A UML-Based Specification Environment[3] introduced in [10]. This system allows us to create models consisting of UML represented in text format with OCL constraints. The original aim of the system was to check if a given instance of the model (or a 'snapshot' in USE terms) satisfies the constraints. In the first versions the user had to create instances manually.

Here we will demonstrate an example. A sample model file is shown in Figure 2.1. This model then can be opened in a system (see Figure 2.2) and manipulated via either a graphical or a command line interface. At any point the current snapshot (state of the objects) can be verified. Figure 2.3 shows the process of work with the model. First an elephant and a trunk objects are created (command !create) and linked together (command !insert). Then the snapshot is verified successfully. After this a new elephant is added with the same name and the snapshot is checked again. This check shows two errors - an error of multiplicity because the new elephant is not related to any trunk, and a violation of the invariant UniqueName.

In the beginning the USE system did not have any functionality for automated instance generation, but later some work has been done in this field. In [11] the authors describe an approach where instances are generated automatically based on the procedures in a special ASSL language (A Snapshot Sequence Language). These procedures specify the bounds and rules for generating objects and linking them as well as setting attribute values. The generation mechanism was based on enumerating all possible combinations.

However, in [12] the ASSL approach is claimed to be ineffective: 'The main disadvantage of the built-in generator is its enumerative nature, as it has to create and check each snapshot if the procedure's snapshot space does not include a snapshot fulfilling all model constraints. As a consequence, larger snapshot spaces which for instance comprise more than a few objects

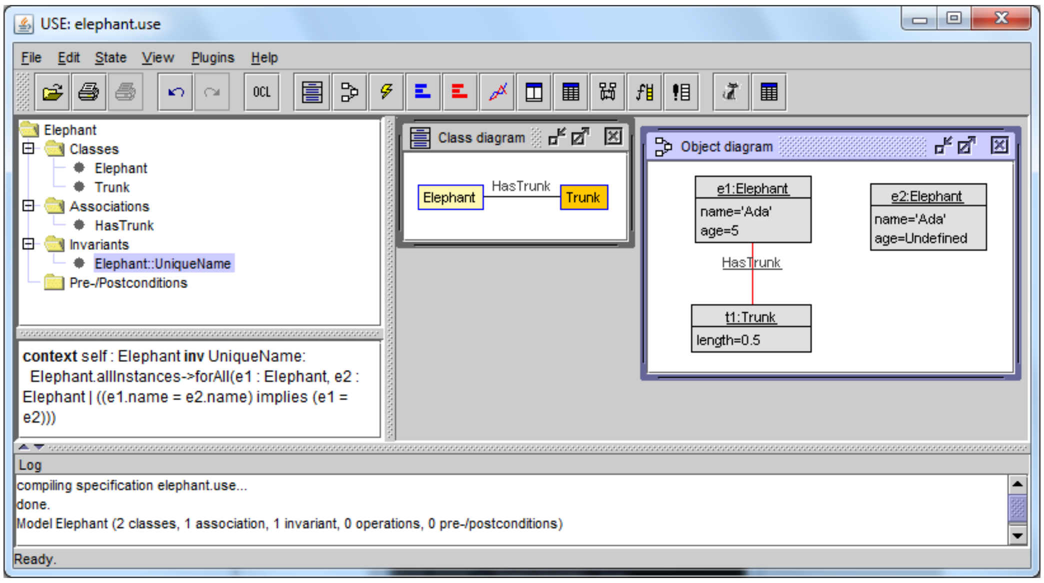
[3]`http://sourceforge.net/apps/mediawiki/useocl/`

```
-- classes

model Elephant
attributes
        name : String
        age : Integer
end

class Trunk
attributes
        length : Real
end

-- associations

association HasTrunk between
        Elephant[1]
        Trunk[1]
end

association InSameFamily between
        Elephant[*]
        Elephant[*]
end

-- OCL constraints

constraints

context Elephant
        inv UniqueName:
                Elephant.allInstances->
                forAll(e1, e2 | e1.name = e2.name implies e1 = e2)
```

Figure 2.1: Example of UML model in USE syntax

and attribute values or all possible link constellations cannot be handled'. A new instance generating mechanism for USE is presented in [12, 13] involving the Kodkod finder which is a SAT-based system specially designed finding instances of models expressed in relational logic (see [14]). Kodkod is also used as an instance generator in Alloy since version 4.0.

The instance generation functionality is available as a plugin for USE called 'Model Validator'. It is currently under active development and not much information on its capabilities is published. Apart from [12, 13] which cover the theoretical aspects of model transition to and from Kodkod rela-



```
context self : Elephant inv UniqueName:
  Elephant.allInstances->forAll(e1 : Elephant, e2 :
Elephant | ((e1.name = e2.name) implies (e1 =
e2)))
```

```
compiling specification elephant.use...
done.
Model Elephant (2 classes, 1 association, 1 invariant, 0 operations, 0 pre-/postconditions)
```

Figure 2.2: USE graphical interface

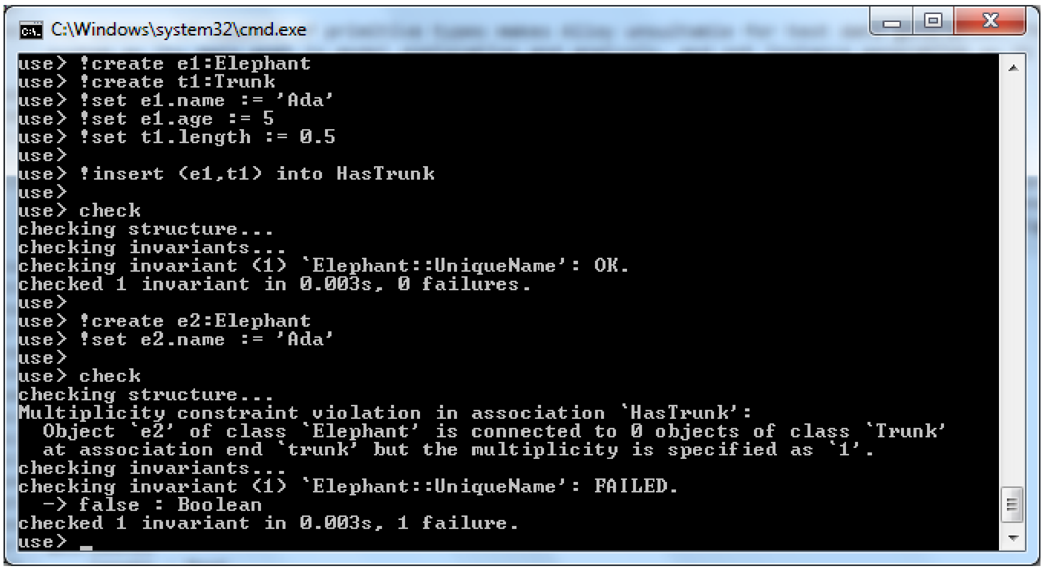
tional logic, the only source of information is the project page on sourceforge[4]. Some documentation can be found in the plugin distribution and in discussion forums such as [15] which gives an insight on how the model finder can be used and what means we have to control the generation process.

The Model Validator plugin has some interesting features and, unlike Alloy, offers some control of the attribute values in the instance. Similarly to Alloy there is a possibility to set lower and upper bounds on the number of classes and relations. Additionally, the user can set the bounds on integers and real numbers. The same bounds will apply to any integer attribute in any class. As for the strings, there are two options. The first option is to set the total number of strings that will be used in the generated instance. In this case strings will have values string1, string2 etc. The second option is to specify the set of strings. All the string attributes will get a random value from this set. A flaw of this scheme is that there is no way to set different ranges of values for different attributes.

The second important point we would like to emphasize is that the user can not only specify bounds for classes, but can also specify a set of object names that will be created. In the example below, the Company class has numeric bounds and the Employee class has a set of objects.

```
Company_min = 3
```

[4]`http://sourceforge.net/projects/useocl/files/Plugins/ModelValidator/`



```
use> !create e1:Elephant
use> !create t1:Trunk
use> !set e1.name := 'Ada'
use> !set e1.age := 5
use> !set t1.length := 0.5
use>
use> !insert (e1,t1) into HasTrunk
use>
use> check
checking structure...
checking invariants...
checking invariant (1) `Elephant::UniqueName': OK.
checked 1 invariant in 0.003s, 0 failures.
use>
use> !create e2:Elephant
use> !set e2.name := 'Ada'
use>
use> check
checking structure...
Multiplicity constraint violation in association `HasTrunk':
  Object `e2' of class `Elephant' is connected to 0 objects of class `Trunk'
  at association end `trunk' but the multiplicity is specified as `1'.
checking invariants...
checking invariant (1) `Elephant::UniqueName': FAILED.
  -> false : Boolean
checked 1 invariant in 0.003s, 1 failure.
use> _
```

Figure 2.3: USE command line interface

```
Company_max = 10

Employee = Set{peter, william, samuel}
```

The purpose of this notation is to enable predefined links between objects. Using these object names it is possible to set the relations between specific objects:

```
Company = Set{HP, Intel}

Employee = Set{peter, william, samuel}

WorksIn = Set{(peter,HP),(samuel,HP)}
```

As we can see, the user has a control over the objects and their relations, but cannot specify attribute values.

Finally, the Model Validator has a feature which is called Automatic Diagram Extraction and means that the user can build a part of a snapshot manually and then use the generation engine to turn it into a full instance. Good thing about it is that user has full control over his part of an instance. However, it is impossible to refer to manually created objects in the bounds specification script. So we cannot set predefined links between manually created and generated objects. All such links will be generated automatically.

2.2.3 ASMIG

ASMIG (A Small Metamodel Instance Generator) is an SMT-based system for generating instances which is being developed in National University of Ireland, Maynooth. The underlying theory is discussed in [16]. The work flow of the system is shown on the figure 2.4.

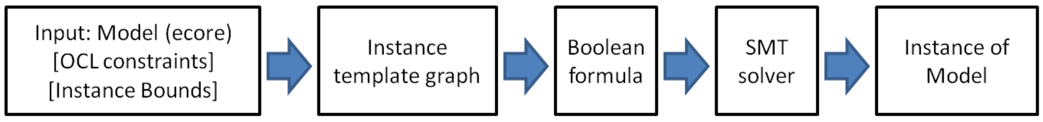


Figure 2.4: ASMIG work flow

The input for the system consists of three parts - a model (it can be a metamodel), optionally some OCL constraints and the bounds for generated instances. The format used for specifying a model is Ecore. It is a format used in Eclipse Modeling Framework (EMF) which is described in detail in [17]. An Ecore file is an XML file specifying the model in terms of Classifiers, Attributes and References. An example of an Ecore model is shown on figure 2.5 (some details are omitted for the sake of clarity). The equivalent diagram is shown in Figure 2.6.

As we can see, this model is a simple metamodel defining an abstract entity Classifier with two attributes and a self-relation (parents) with multiplicity many-to-many. There is also an entity Class, a sub-type of Classifier related to zero or more Properties and Operations.

The OCL constraints can also be specified in a separate text file. As for the instance bounds, there is no easy way of specifying those as the corresponding user interface hasn't been created yet, but it is possible to configure the bounds in the source code using API. There are two kinds of bounds in the system - bounds on the number of objects for a specified entity and bounds on the lengths of string attributes. The default values are 3 for number of entities and 5 for string length.

Examples of generated instances for this input model are shown in Figures 2.7, 2.8, 2.9. The bounds in this case are set to 2 for number of instances and 2 for length of strings. The orange nodes in the diagrams represent objects while gray nodes are attribute values. The integer values are not bounded explicitly, so the bound is their natural range.

Now after giving this example let us examine how the generation process in ASMIG is performed (see Figure 2.4 again).

The first step is generation of the instance template graph (ITG). The graph for this example is shown in Figure 2.10. After the model is read the

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0">
  <eClassifiers type="EClass" name="Classifier" abstract="true">
    <eStructuralFeatures type="EReference"
        name="parents"
        upperBound="-1"
        eType="#//Classifier"/>
    <eStructuralFeatures type="EAttribute"
        name="name"
        eType="EDataType EString"/>
    <eStructuralFeatures type="EAttribute"
        name="hash"
        eType="EDataType EInt"/>
  </eClassifiers>
  <eClassifiers type="EClass"
    name="Class" eSuperTypes="#//Classifier">
    <eStructuralFeatures type="EReference"
        name="ownedAttribute"
        upperBound="-1"
        eType="#//Property"/>
    <eStructuralFeatures type="EReference"
        name="ownedOperation"
        upperBound="-1"
        eType="#//Operation"/>
    <eStructuralFeatures type="EAttribute"
        name="isAbstract"
        eType="EDataType EBoolean"/>
  </eClassifiers>
  <eClassifiers type="EClass" name="Property"/>
  <eClassifiers type="EClass" name="Operation"/>
</ecore:EPackage>
```

Figure 2.5: Sample Ecore model

graph is generated in the following way. For every non-abstract classifier in the model the nodes are created according to the bound of this classifier. As the bound for all classifiers is 2 in our example there are two instances created for Class, Operation and Property entities. For the Classifier entity no nodes are created as it is abstract. In this simple example everything is clear, but in case of inheritance hierarchies a special algorithm is applied - the number of nodes (which is equal to bound) is distributed across all hierarchy so that the number of nodes for all subclasses sums to the value of the bound. In case there are more members in the hierarchy then a bound, one node is created for every subclass.

Then for every object node a number of attribute nodes are created

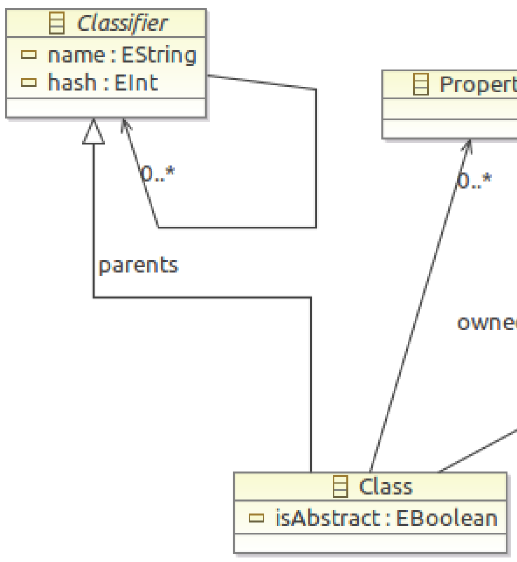


Figure 2.6: Ecore model diagram

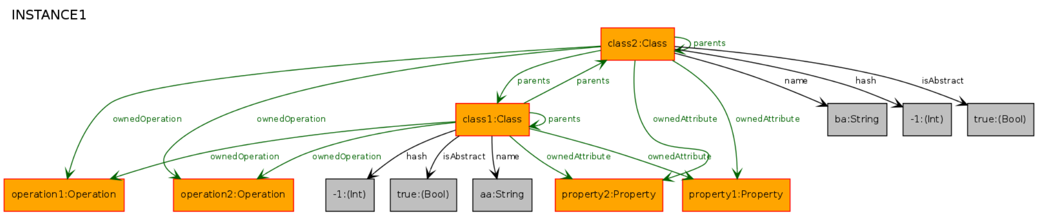


Figure 2.7: Example instance 1

(shown as rectangles).

Finally, for each relation in the model between classifiers A and B, edges in the graph are created between all nodes of type A (and its subtypes) on one side and all nodes of type B (and subtypes) on the other side. In other words, everything that can be connected according to the model specification gets connected.

The next stage is translating the graph into a boolean formula. The presence of every node in the instance is encoded in a boolean variable. Every edge of the graph is also represented as a boolean variable. So in the resulting instance every node and edge from the ITG may be 'switched on' which means the node or edge will appear in the instance or 'switched off' which means the opposite. Here OCL constraints also come into play. ASMIG code for translating OCL partially reuses code from the USE system

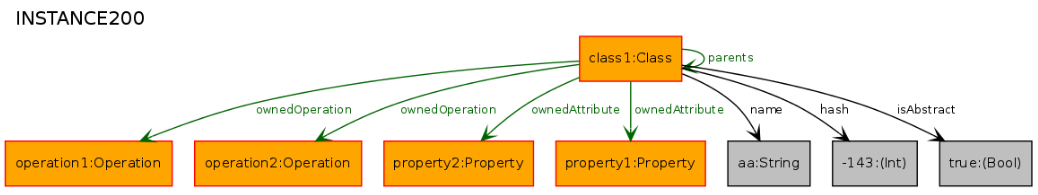


Figure 2.8: Example instance 2

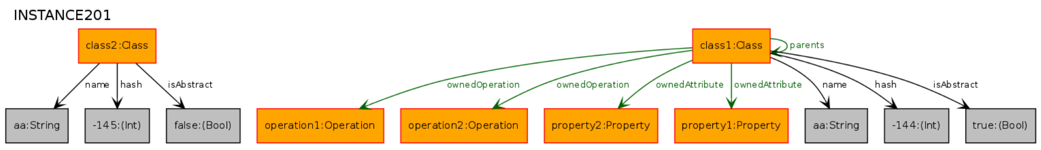


Figure 2.9: Example instance 3

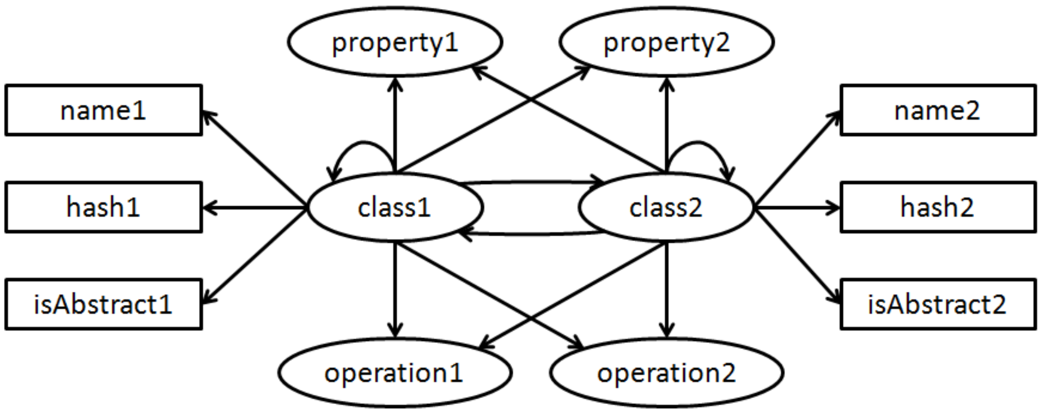


Figure 2.10: Instance template graph

(Subsection 2.2.2) which is open-source.

The attribute values are also encoded in the formula. Here we would like to draw reader's attention to the way string values are instantiated in ASMIG. As we saw, the USE system allows us to predefine a set of strings that will be used in the instance or specify the number of strings. Unlike this, in ASMIG the strings are actually generated. It is accomplished by treating each character in a string separately. For every character a boolean expression is created stating that the character can be one of the set of possible characters. That is why the bound on the string length has to be specified. Usually the generated strings look like ‘aaaaa’ because ‘a’ is a first symbol in the range of permitted symbols.

This mechanism of string generation gives a benefit of using string-related OCL constraints. Let’s consider the following example. In a transaction processing system the transaction can be either incoming or outgoing. Each transaction has a text code which is a string consisting of a letter I for incoming transactions or O for outgoing, followed by transaction number. The incoming transactions have flag In set to true. A model designer might specify the following invariant:

```
context Transaction
    inv validCode:
        if In then
            code.substring(1,1) = 'I'
        else
            code.substring(1,1) = 'O'
        endif
```

ASMIG is designed to be capable of generating strings that satisfy such invariants. Both the invariant and the string (character-wise) will be translated to boolean expressions and the satisfaction of this expression will give the string starting with either I or O. As the part of system responsible for translating OCL is still under development and not ready for usage yet we cannot illustrate this example with the Transaction instances generated by ASMIG. However, we believe it will be possible when the development is over.

Additionally, ASMIG provides generation for attributes with Enum type which we haven’t seen in other systems. Enum values are handled as integer values bounded by the number of Enum elements.

The boolean formula generated from the instance graph and the OCL constraints is then solved by the Z3 theorem prover [18] which is a fast SMT solver developed in Microsoft Research. Z3 has native support for integers which is valuable for ASMIG as we don’t need to specify bit-width and translate integers into sets of booleans (as is done in Alloy).

Finally, the values obtained from Z3 that represent an instance are mapped back to the nodes and edges of template graph. The result is expressed in the DOT language and rendered into an image with Graphviz.

To generate the next instance of the model, the boolean formula is modified in the following way:

$$NewFormula = (OldFormula) \wedge \neg (Var_1 = InstVal_1 \wedge Var_2 = InstVal_2 \wedge ...)$$

Here Var_i is a variable that defines some aspect of an instance (e.g. presence of a node or edge or attribute value). $InstVal_i$ is a value for this variable assigned by Z3. This modification is necessary to ensure all instances are unique.

Following our way of examining model finders, let's summarize the means the user has to influence the generated instances in ASMIG. Firstly, it is possible to specify the bounds on the number of objects in the instance for every classifier separately. Secondly, the length of string fields can be bounded. There is no way to set attribute values or somehow influence relations between objects.

As we can see, ASMIG provides about the same level of control over the instance generation as Alloy does, while USE has has more to offer in this respect.

The whole idea of current work, however, is to enhance ASMIG system with an instrument to edit the instance template graph therefore allowing users to predefine nodes, relations and, what is most important, attribute values of the generated objects. The proposed solution is to design a domain-specific language for defining the template graph. This will make the system truly helpful in the task of generating test data sets, which is one of the purposes of instance generation.

The principal schema of how we are proposing to improve ASMIG is shown in Figure 2.11. Here and later to avoid confusion we will call this improved system ASMIG+.

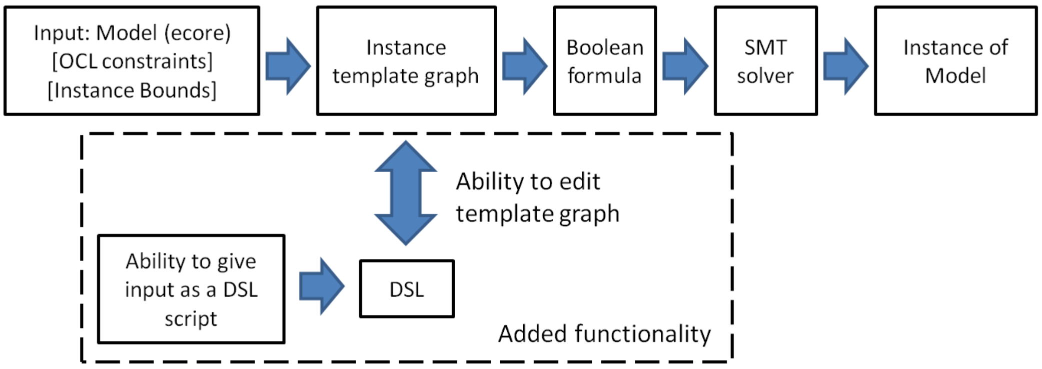


Figure 2.11: ASMIG+ work flow. The new functionality enables the user to interact with the ITG directly

2.2.4 Comparison of instance generation systems

The summary of the characteristics of described systems is shown in Table 2.1. As can be seen, the USE system has the richest support for instance generation configuration among the existing systems. However, our ASMIG+ system is supposed to give the user even more control over the instance generation.

2.3 Domain-Specific Languages

In this section we will give an overview of domain-specific language (DSL) engineering. This area is directly connected with the current project as the proposed approach for adding new functionality to ASMIG system is to create a special language for defining and editing instance template graphs that were previously hidden from the user.

To start with, we will give a definition of a DSL by Martin Fowler [19, p. 27]: 'Domain-specific language (noun): a computer programming language of limited expressiveness focused on a particular domain'. The two key features given in this definition which are in a way consequences of each other are the limited expressiveness and focus on a particular domain (or to be more precise usually on a particular task). Limited expressiveness means that in DSLs we normally find a small set of operators as opposed to general purpose languages with a wide range of constructs and operators serving different needs. As a rule, DSLs are not Turing-complete. The focus on a specific problem means that a DSL is designed to serve one purpose in a particular domain.

DSLs are not something rare or new, and we may use some of them regularly, probably without thinking of them as DSLs. For example, all following languages fall into the category of domain-specific languages: SQL, Graphviz/DOT, LINQ, CSS, ant. Two things are common for all the languages in this list - they have limited expressiveness and they are aimed at one particular task.

The main benefit of using a DSL is that it is small and much easier to learn then a general purpose language. Because of their focus on one task, DSLs are very clear and expressive. This has two consequences. First of all, using a DSL can considerably increase engineering productivity. Consider the difference between using SQL and retrieving data from the DBMS programmatically using the API. As a result it is easier to maintain the code written in such languages. Secondly, a DSL can be easily understood not only by software engineers, but also by domain experts, analysts etc. This sec-

	Alloy	USE	ASMIG	ASMIG+
Model format	Own language	UML	Ecore	
Constraints format	Own language	OCL	OCL	
Instance generation	automatic	manual automatic mixed	automatic	automatic preedited
Class boundaries	Yes	Yes	Yes	Yes
Reference boundaries	Yes	Yes	No	Yes, by limiting ITG
Integer generation	Yes, bounded by bit-width	Yes, bounded by range	Yes, unbounded	
Float generation	No	Yes, bounded by range and step	No	
String generation	No	Predefined set of strings	Yes, bounded by length	
Enum generation	No	No	Yes	
Generation from partially created instance	No	Yes	No	Yes, by editing ITG
Specifying attribute values on object level	No	Yes, via ASSL (inappropriate for large instances)	No	Yes

Table 2.1: Comparison of characteristics of instance generating systems

ond point is more beneficial in the industrial cases than in research projects. However, a good DSL can help to communicate the domain and the problem to people who are not experts in a certain area.

As for the mechanics, the processing of a DSL is quite similar to the processing of a general purpose programming language. Figure 2.12 shows the flow of DSL processing.

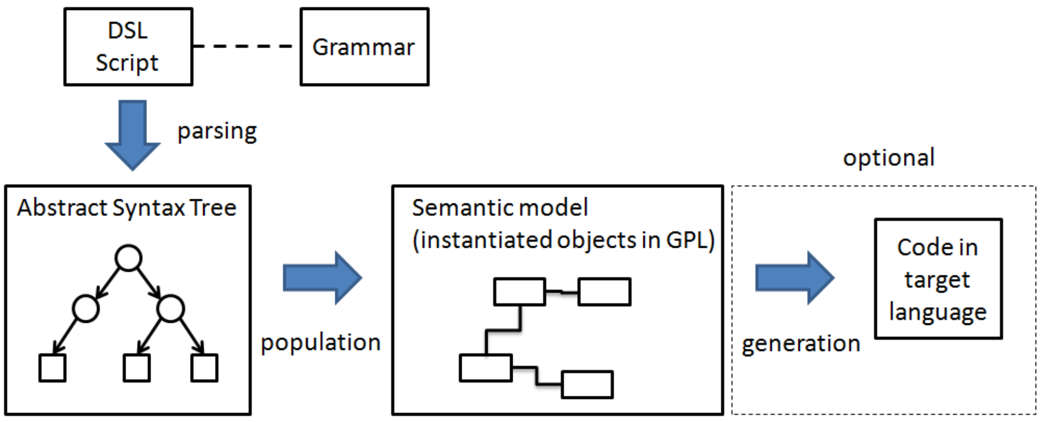


Figure 2.12: DSL Mechanics

The script conforming to the grammar of the language is scanned and parsed using standard techniques and the result is an abstract syntax tree. The tree is then traversed and used to populate the semantic model. A semantic model is a fragment of system consisting of classes that contain the data which the DSL is meant to express. A DSL can be seen as a front-end to the semantic model. For example, in the DSL for instance template graphs in ASMIG, the semantic model will be a part of ASMIG that stores this graph. Some DSLs are created to simplify code generation. In this case the next step of the processing is the generation of code in a target language.

There are two types of DSLs - Internal and External. Internal DSLs are built on the top of the general purpose (host) languages. These languages should conform to the syntax of a host language. An example of internal DSL is the LINQ language (Language-Integrated Query) for querying data storages. A typical approach used in internal DSLs is method-chaining. Figure 2.13 shows an expression in LINQ that illustrates this concept of ‘language inside of a language’.

The host language for LINQ is C#. However, the code in Figure 2.13 does not look like typical C# code. It has its own structure and rules that are built on top of C# syntax.

External DSLs on the contrary have their own syntax and are parsed separately according to Figure 2.12. In our current work we will be talking about an external DSL.

Although domain-specific languages have been used in the industry for a long time, we can see that the interest to this area is increasing nowadays.

```
var weakestUnitId = units.Select(x => x.Id)
    .Where(x => x.Player == p1)
    .OrderByDescending(x => x.Health)
    .ThenBy(x => x.Level)
    .First();
```

Figure 2.13: LINQ code example

The reason for this is the appearance of so-called ‘language workbenches’ - special systems for creating languages. Among these systems are Spoofax, MPS, Xtext, EMFText and many others. These tools simplify defining the grammar of the language, the syntax, binding the semantic model, defining editing behaviour and so on. A detailed overview of language workbenches and their capabilities can be found in [20].

For further reading on the subject of DSL engineering we may recommend books [1] by Anneke Kleppe giving the theoretical background and a big picture, [19] by Martin Fowler with many practical advices and a reference list of applicable patterns and [21] by Markus Voelter which focuses on implementing DSLs using different language workbenches.

2.3.1 Why we are using EMFText

The language workbenches available now provide more or less similar functionality. Hence it is not easy to argue for one variant over another. There are many papers comparing different workbenches and pointing out the differences and similarities [22], [23], [24], but there is no significant distinction between them.

In our case, however, the choice is narrowed by the fact that the ASMIG system is already interacting with Eclipse Modeling Framework. Therefore it is more convenient to use one of the Eclipse-based workbenches - EMFText [25] or Xtext. There is a comparison between these two systems [26] which tells us that Xtext is more language-oriented, while EMFText follows a model driven approach.

Furthermore, when using EMFText the language metamodel plays a central role (see [27]) and it is specified in the EMF Ecore format discussed previously in Subsection 2.2.3. This means we can experiment with the instance generation for our new language using ASMIG. This fact decides our choice in favour of EMFText.

Ideally after the creation of the language we will be able to generate instances of our DSL metamodel (in other words, scripts in our DSL) which we can then feed into ASMIG as test input and therefore prove the whole concept of using instance generator for generating test data sets.

2.3.2 Graph definition languages

When there is a task of creating a language one is supposed to use the experience of previously created languages in the same area. Our task of defining instance template graphs for model finding is of course too narrow to expect the existence of languages for this. However, in the more general problem of graph definition there are several existing languages which are listed below:

- GXL (Graph eXchange Language).
- Graph Modeling Language (GML).
- Directed Graph Markup Language (DGML).
- Graph ML.
- Trivial Graph Format.
- DOT language.

Among the these languages GXL, DGML and Graph ML are XML-based. We are not interested in the XML-based languages as we want our own instance template graph language to be more human-friendly than computer-friendly.

Trivial Graph Format is a simplistic language that specifies only nodes and edges with labels:

```
1 First Node
2 Second Node
#
1 2 I am an edge
```

The GML represents a graph in a following manner (fragment of an example from [28]):

```
graph [
  comment "This is a sample graph"
  directed 1
  IsPlanar 1
  node [
    id 1
    label "Node 1"
  ]
  node [
    id 2
    label "Node 2"
  ]
  edge [
    source 1
    target 2
    label "Edge from node 1 to node 2"
  ]
]
```

This language has many keywords that in our opinion can be removed from the language without any harm. So we would not like to use it as an example for our language.

DOT is a domain-specific language for the Graphviz system introduced in [29] and designed for describing graphs for two-dimensional graphical visualisation. In the ASMIG system Graphviz is used to render resulting instances (for example, Figure 2.7).

DOT is a rich language able to express many aspects of graph visualization - shape and colour of nodes and edges, labels, subgraphs, composite nodes and many more, for details see [30]. An example of graph definition and rendered visualisation is shown in Figure 2.14.

We can see that nodes and edges may have different attributes, and the edges are defined with the construct '->' which is very simple and expressive.

Although the task of DOT is different from the task of our ASMIG DSL, there are certain similarities. In our DSL the user also should be able to define a graph and to set node attributes. On the other hand, we would like to know the type of the object represented by a node to check the correctness of relations which is absent in DOT. In spite of these differences we can consider reusing some subset of DOT syntax for our DSL.

```
//example of directed graph
digraph Example{

//nodes
  salt [fillcolor="gray",style="filled"];
  p [label="Buckwheat porridge",shape=square];
  butter

//edges
  salt -> p [label = "1 sp", style="dashed"]
  butter -> p
}
```

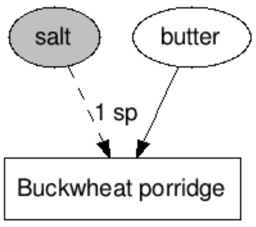


Figure 2.14: DOT and Graphviz example. Graphviz generates the graph image from the graph definition in DOT language

Chapter 3

Solution

3.1 System design overview

Here we will discuss what components have to be added to ASMIG system in order to turn it into ASMIG+ with the precise control over the instance generation. Figure 3.1 shows the principal schema of the new modules and their interaction with existing system.

The new part in the figure is shown in detail whereas in the old part only essential blocks are present. This is why there are more components in the 'new' part of the diagram. It has no relevance to the actual complexity of the systems. The ASMIG system of course is much more complex.

One of the key principles we used in the design is the minimization of impact on the existing ASMIG code. This is done in order to keep the coupling between the two parts as low as possible. Additionally, we did not want to interfere with the development process of ASMIG.

Another key principle we took is testing of the system on every stage which is important in the context of system evaluation.

Figure 3.1 shows what parts are there in the system. In order to understand why we need all these components let us go back to Figure 2.11 and remind ourselves the use cases of the system.

First of all, the user should be able to create an instance template graph (ITG) using our domain-specific language.

This use case is supported by the 'up' arrows in Figure 2.11. The user creates a script which is then converted into a model (this is the way EMFText works). Then a 'DSL-Factory converter' traverses the model and creates a graph in the ASMIG data structure called 'Factory'. This factory does not have an API for creating custom graphs, it is designed to create graphs from Ecore models. In order to manipulate this Factory we have provided an API.

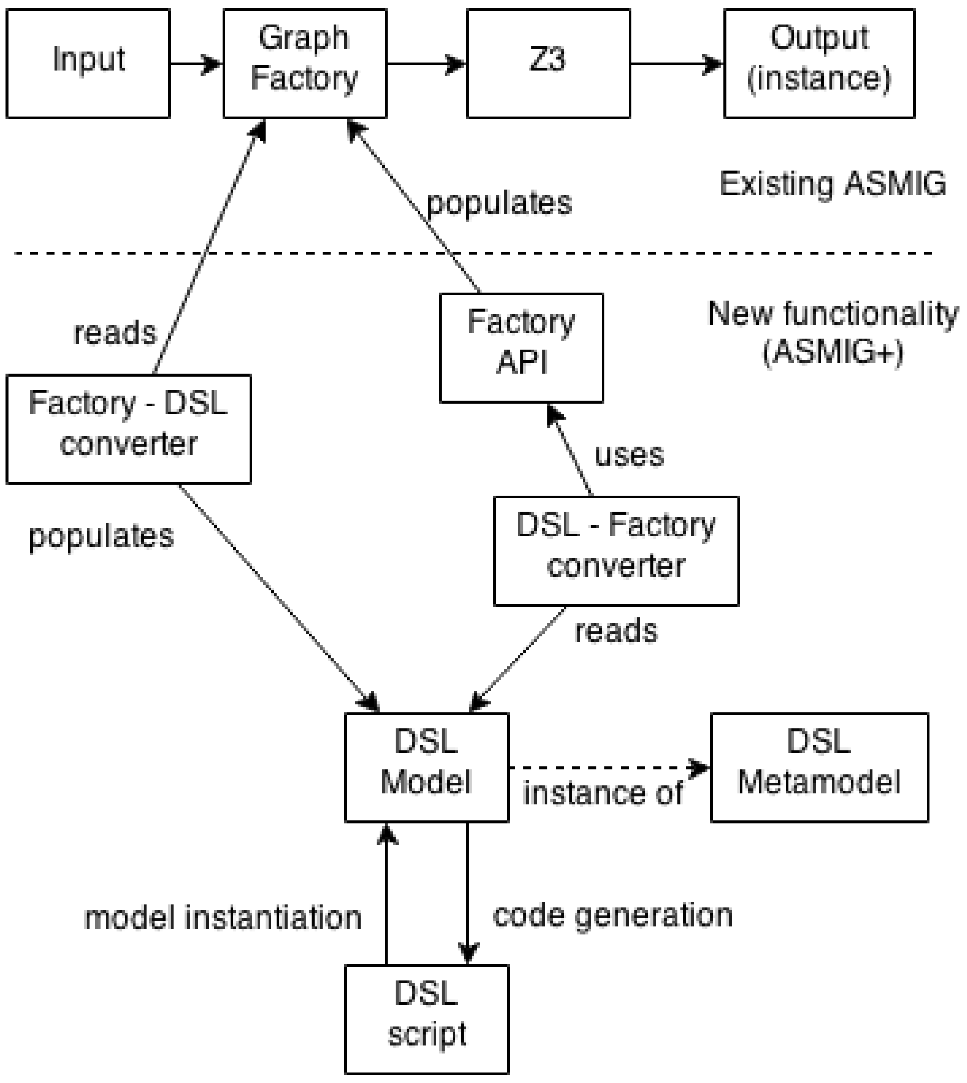


Figure 3.1: ASMIG+ Design overview. The new part of the system (below dashed line) integrates with the existing ASMIG system (above the line) to provide new functionality

This instance template graph is then used to generate instances.

The second use case is editing the graph. The user gives an Ecore model as an input, and then the ITG is generated according to the bounds. After this, the user should be able to modify this graph. Here we need the 'down' arrows of Figure 2.11 as well. The 'Factory-DSL converter' walks the ASMIG graph and populates the DSL Model which is then translated into a script.

We used the incremental process in system development, with iterative cycles of design, implementation and testing. It took us three iterations to build the whole system.

The material of this chapter is structured by iterations because this way logically follows the engineering path, and better reveals the decisions we made and the reasons behind these decisions.

1. In the first iteration we created and tested the innermost layer, a Factory API. This also required slight modifications of ASMIG code.

2. In the second iteration we worked on the DSL Metamodel and the converters between graphs in DSL and ASMIG representations. We also performed integration and system testing.

3. In the third iteration the language based on the DSL Metamodel was implemented using EMFText language workbench. The integration and system testing performed again.

These three iterations are illustrated in Figure 3.2.

Next three chapters describe each iteration in closer detail.

3.2 Iteration 1: Factory API

3.2.1 ASMIG Factory Overview

In order to integrate new functionality into any existing system we should first of all study this system. So our starting point is the code of ASMIG. The system includes 24 packages having in total 238 classes and 16968 lines of code (measured by State Of Flow Eclipse Metrics plugin[1]). At the moment we are only interested in the part of ASMIG that is related to creating and storing instance template graph as our task here is to populate these data structures with our custom graph. A class diagram showing the corresponding fragment of the system is shown in Figure 3.3. This diagram has been simplified and slightly modified for better readability as a representation of the actual ASMIG code.

First of all we would like to discuss the class Factory. This class is important as it contains the data structures representing the graph. The graph is stored in three fields of Factory - BVG, ENA and ST. We are not quite sure why these reference are named this way, but in order to talk about the

[1]`http://www.stateofflow.com/projects/16/eclipsemetrics`

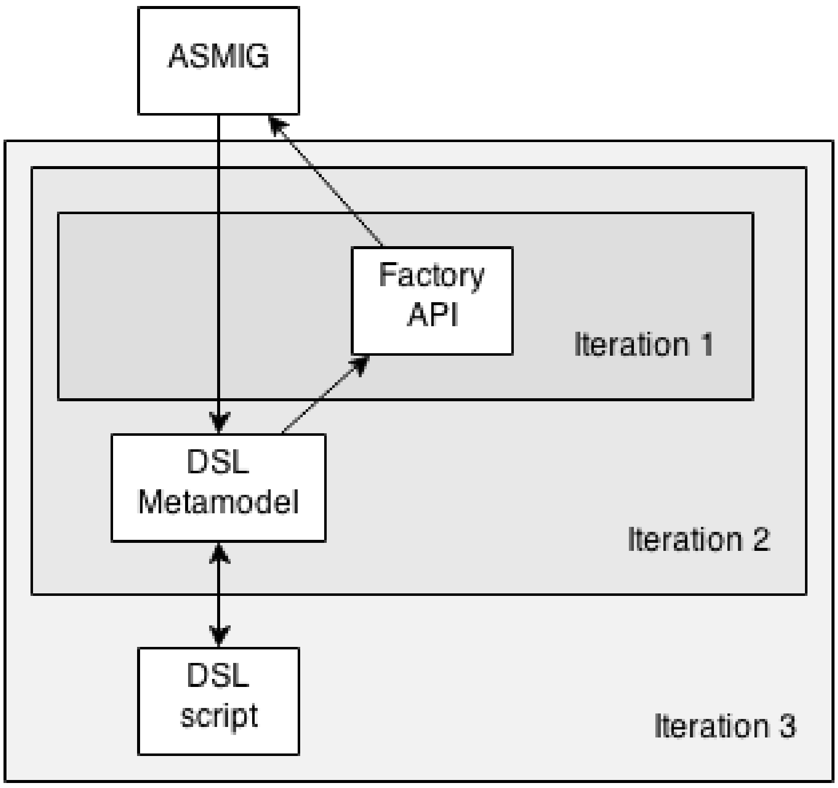


Figure 3.2: ASMIG+ Development iterations

system we will be using its terminology. BVG is a collection of nodes, ST is a collection of edges and ENA is a collection of edges for attributes.

There are two types of nodes in the graph. GraphNodes represent the actual nodes of the ITG (objects that will or will not appear in the generated instance). The second type of a node is DataNode. DataNodes store the information about GraphNode attributes. A DataNode contains a value. Depending on the attribute type this value is one of the subclasses of an abstract Value class. ASMIG supports the generation of values for the following data types: Boolean, Integer, String, Enumeration. If a DataNode has a NULL value the actual value is generated by Z3. Otherwise if the value is set, this value will appear in the instance.

Here we would like to mention that Enumeration values cannot be predefined in the current version of ASMIG. This functionality is not implemented. This means that Enumeration values will always be generated. Integer, Boolean and String values can be set.

So coming back to the Factory class, BVG field stores a collection of GraphNodes, ST is a collection of edges that connect a GraphNode to another

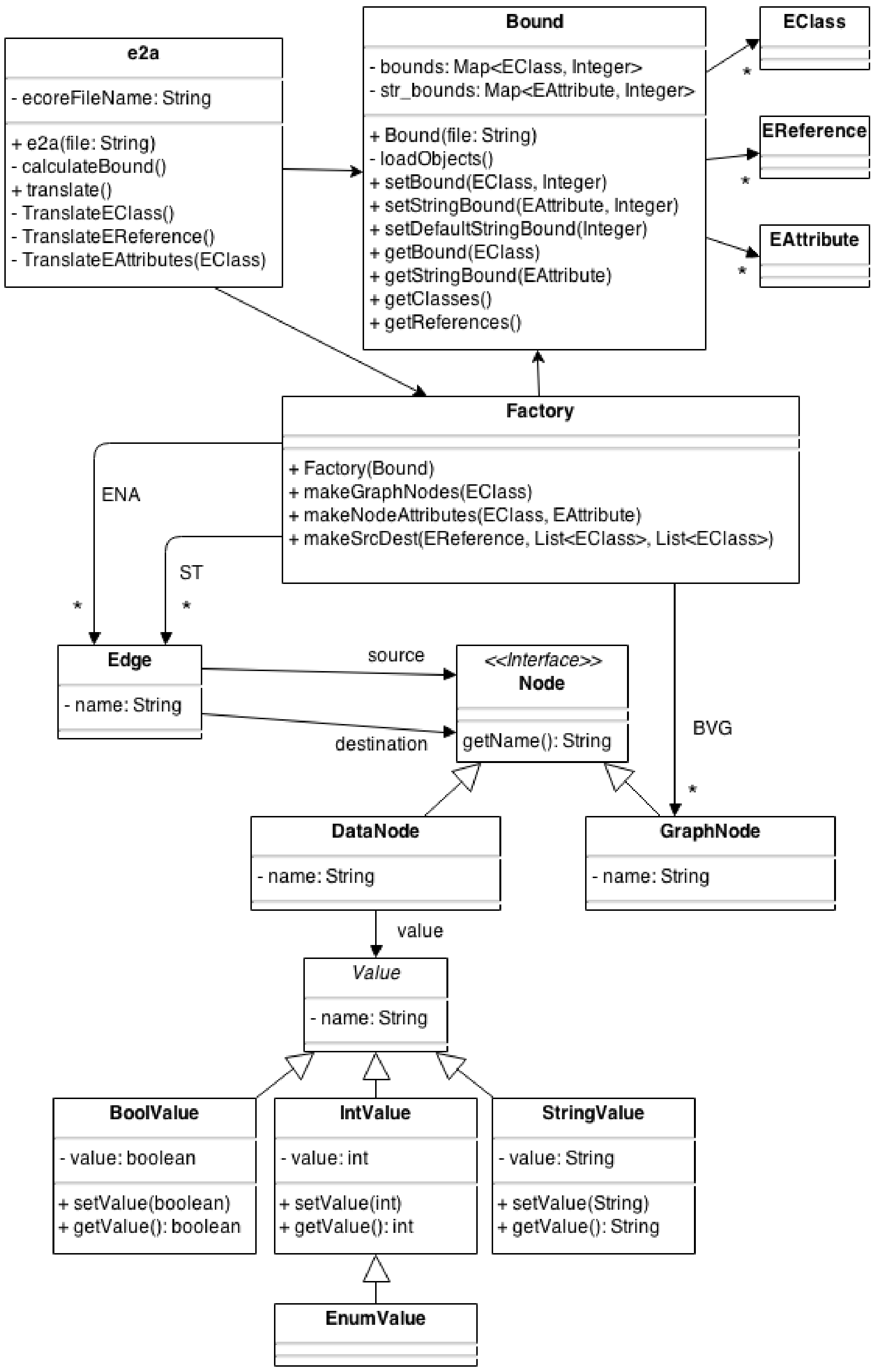


Figure 3.3: Class diagram of ITG-related fragment of ASMIG system

GraphNode. These edges correspond to EReferences of the Ecore model.

Finally, ENA is a collection of edges between a GraphNode and a DataNode. These edges tell us which nodes have what attributes.

Now let us draw our attention to the upper part of the diagram in Figure 3.3 - classes e2a and Bound. e2a (Ecore model to attributed graph) is a class that acts as a coordinator of the whole process of graph creation and population. The class Bound serves two purposes. First of all, it stores information about numeric bounds that limit the instance generation space. Secondly, the Bound class has references to the contents of the Ecore model. This data is available through methods `getClasses()` and `getReferences()`.

The process of creating the ITG can be divided into two phases - initialization and translation. The initialization process is shown in a form of a sequence diagram in Figure 3.4.

We can see that the entry point is an instance of class e2a. This object takes a file name of the Ecore model (*.ecore file) as a constructor parameter. Then a Bound object is created which reads the model (EClass, EReference, EAttribute from the Ecore file). After this a Factory object is instantiated.

The bounds are also calculated in the initialization phase. The method `calculateBound()` sets the user-defined class bound for the classes, applying some additional logic for inheritance hierarchies. This logic should not bother us now. As a result of the bounds calculation the Bound object stores integer bounds for every non-abstract EClass on the model.

Then the default string bound is set. In ASMIG string bound can be set either for the whole model (it is a default string bound) or for a specific EAttribute.

As we can see from the diagram all the initialization is performed in a constructor of e2a. Afterwards this e2a object can be used to perform a translation phase which translates a model into a graph.

The first part of this process (translation of classes) is shown in Figure 3.5. We don't show the whole process as the diagram would be too large and complicated. Moreover, first translation step is enough to give us the impression of how the system works.

The translation happens in the `e2a.translate()` method. This method does not do anything else apart from calling private methods `Translate-EClass()` and `TranslateEReference()`. In `TranslateEClass()` there is a loop which invokes `factory.makeGraphNodes(EClass)` for every EClass in the model. The Factory then gets the bound for this specific class and instantiates corresponding number of GraphNodes. Then control returns back to `TranslateEClass()` method which invokes `TranslateEAttributes(EClass)` for the current class (not shown on the diagram). This method iterates over all EAttributes of the EClass and invokes `factory.makeNodeAttributes`

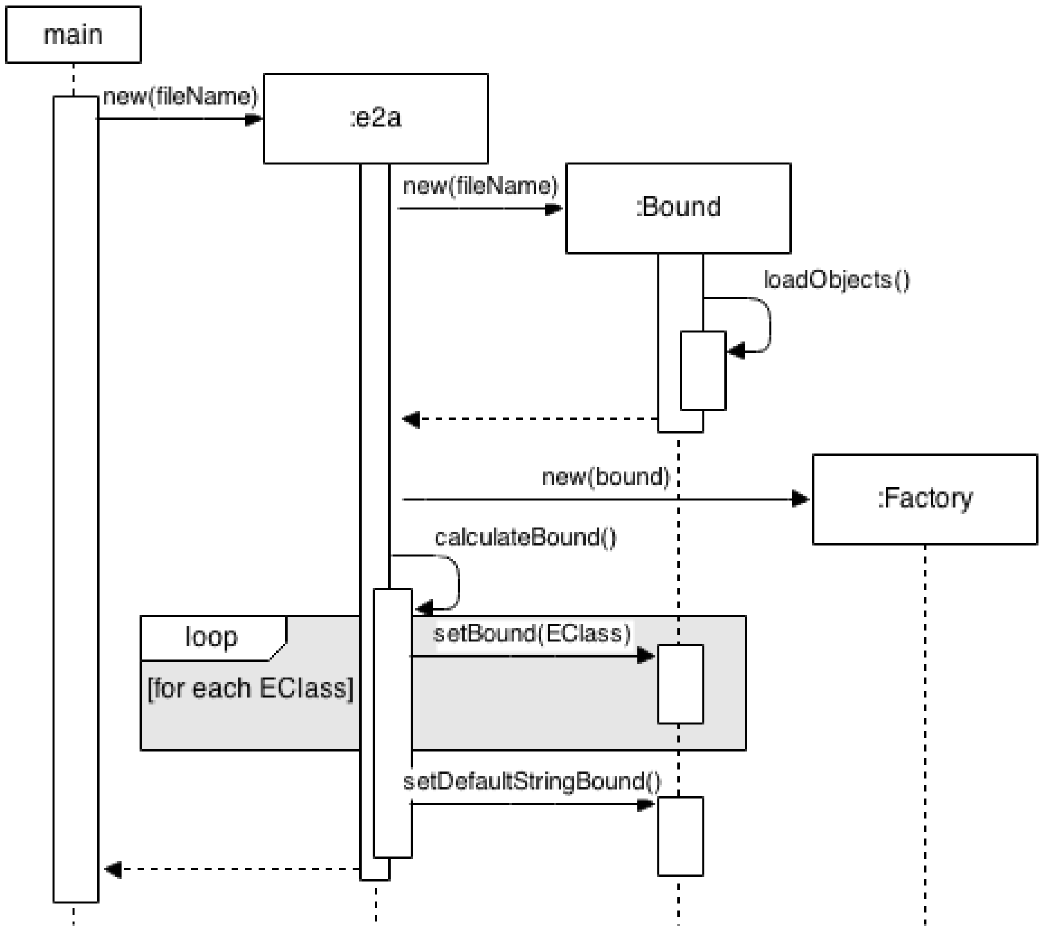


Figure 3.4: Initialization of Factory

`(EClass, EAttribute)` which instantiates DataNodes and Edges connecting DataNodes to previously created GraphNodes.

Therefore by the end of the method `e2a.TranslateEClass()` factory stores a collection of nodes with attributes (BVG and ENA collections). Then `TranslateEReference()` method iterates over all EReferences in the model and calls `makeSrcDest(EReference, List <EClass>, List<EClass>)` in `factory` object which populates ST collection. We would like to explain why here a list of classes is passed instead of one class. This takes into consideration the references between subclasses. For example, if a reference e connects classes A and B, than e can also connect any subclass of A with any subclass of B.

After the e2a object is created and the `translate()` method is invoked

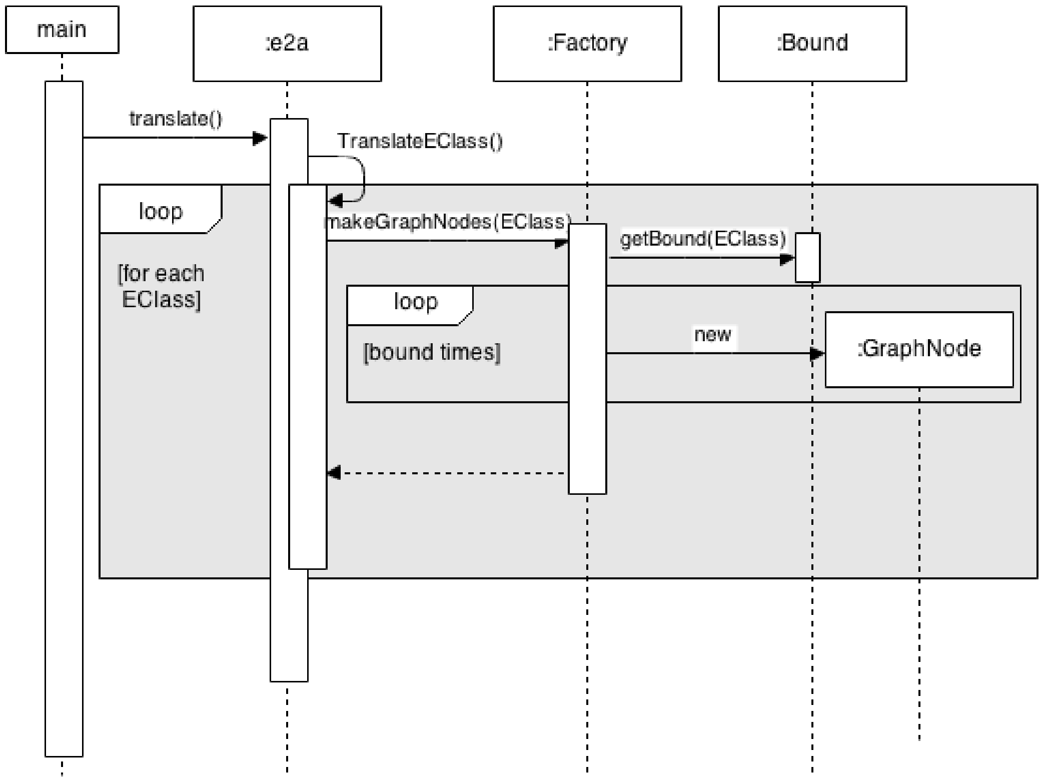


Figure 3.5: Population of Factory (fragment)

this e2a object is passed on for further processing.

To summarize, the Factory object is created in the e2a constructor and the ITG is populated using these methods:

```
makeGraphNodes(EClass)
makeNodeAttributes(EClass, EAttribute)
makeSrcDest(EReference, List<EClass>, List<EClass>)
```

This brings us to two important conclusions:

1. We need to have an Ecore file in order to instantiate e2a and a Factory.
2. We cannot create an arbitrary graph using the current functionality.

The first point is understandable and logical. We will need a model anyway to check that the graph is valid, so it is not a problem.

The second aspect means that we have to implement our own mechanism for creating a graph inside a factory.

3.2.2 API Design

We need the API for a Factory which would allow us to perform the following operations:

1. Create a node of some type.
2. Create edge between two nodes.
3. Set node attribute value.

We would like our nodes to have name and a type. A type is one of EClasses from the model. We would also like to prevent creating invalid graphs - e.g. with types that are not in the model or with edges between objects that don't have a reference in a model.

The proposed design of a solution that fulfills these requirements is shown in Figure 3.6. We decided to use a pattern Facade from a 'Gang of Four' book [31]. The reasons for using this was our wish to provide a clean API for creating graphs and isolate all the complexity of ASMIG.

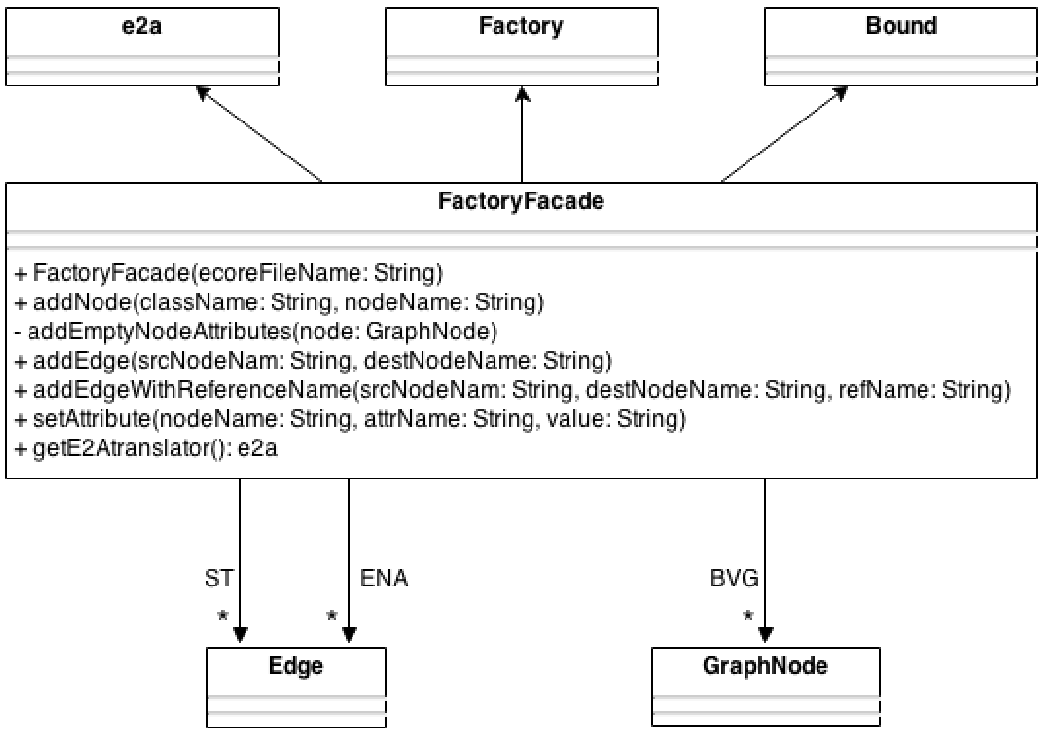

e2a
Factory
Bound
FactoryFacade
+ FactoryFacade(ecoreFileName: String)
+ addNode(className: String, nodeName: String)
- addEmptyNodeAttributes(node: GraphNode)
+ addEdge(srcNodeNam: String, destNodeName: String)
+ addEdgeWithReferenceName(srcNodeNam: String, destNodeName: String, refName: String)
+ setAttribute(nodeName: String, attrName: String, value: String)
+ getE2Atranslator(): e2a
ST
ENA
BVG
*
*
*
Edge
GraphNode

Figure 3.6: FactoryFacade class diagram

The FacadeFactory holds references to the e2a class, Factory class and Bound class. Moreover, it also holds references to the BVG, ST, ENA collections to be able to create the graph directly. It may seem strange that we need a reference to the Bound class since we don't need the bounds. However, this class stores not only the bounds, but the whole Ecore model as well (EClasses, EReferences, EAttributes). We need this information to check that our custom graph conforms to the model.

The work flow for the FactoryFacade consists of three stages - creation, creating an arbitrary graph using API methods and finally call to a method `getE2Atranslator()` which returns a e2a object which then can be used in the following steps of ASMIG work flow.

The instantiation of FactoryFacade is shown in Figure 3.7. The approach can be argued as breaking Factory encapsulation. However, the Factory class already contains methods `getBVG()`, `getST()` and `getENA()` which are used at some later stage to read the graph. So the graph is not really encapsulated inside the factory.

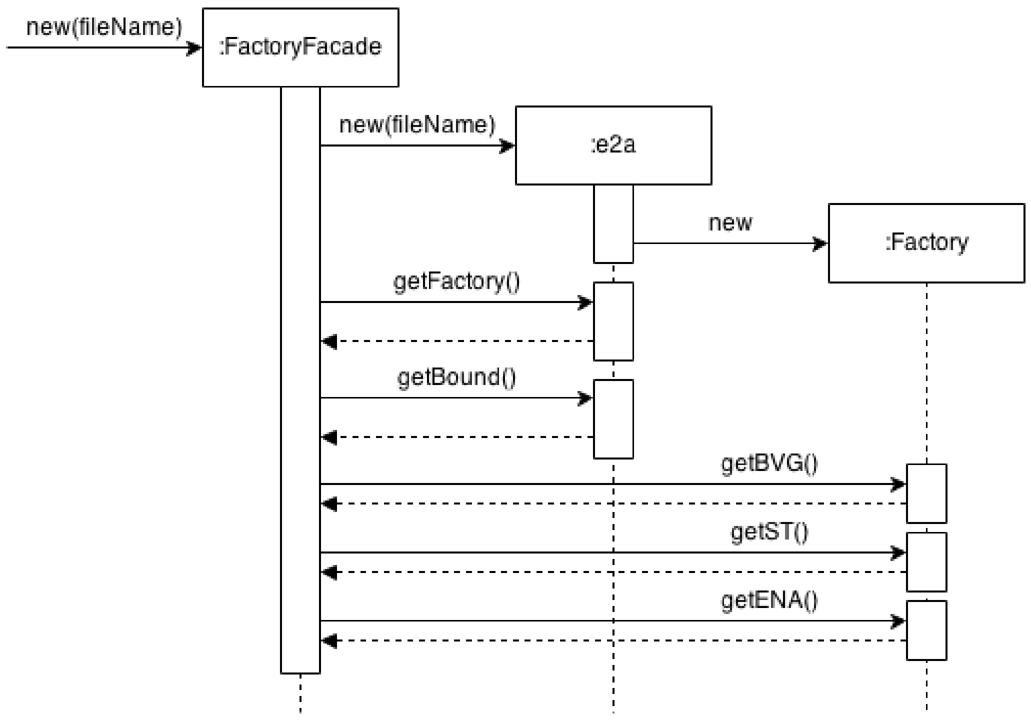


Figure 3.7: FactoryFacade instantiation sequence diagram

FactoryFacade class also performs the validation of a graph in the terms of conformance to the model.

Now let us examine the API methods:

```
addNode(className: String, nodeName: String)
setAttribute(nodeName: String, attrName: String, value: String)
addEdge(srcNodeNam: String, destNodeName: String)
addEdgeWithReferenceName
    (srcNodeNam: String, destNodeName: String, refName: String)
```

All the input parameters are passed in the form of strings because the client of FactoryFacade will get the node names, class names and all the rest from the text script and there will be no more appropriate place to perform this validation.

The method `addNode()` not only adds the node to the graph, but also adds all the DataNodes for all attributes this node may have. The attributes have empty values which means that by default the attributes will get values generated by Z3 (as it is in the existing version of ASMIG). Method `setAttribute()` allows to set an exact value for an attribute and it means that this value will appear in the resulting instance without changes (of course if the node will be 'switched on' in the instance).

As for the edges, we have two methods. Method `addEdge()` adds an edge between two nodes if there is only one reference between classes of these nodes, and `addEdgeWithReferenceName()` adds an edge between two nodes for a specific reference. It is possible that two classes have more than one link between them. For example, if we have a bidirectional linked list we will normally have a class Node with fields next and previous referencing Node objects. In this example the Ecore model will consist of one EClass (node) and two EReferences connecting Node to itself.

We could leave only one method which takes reference name, but we have a reason to keep two methods. For the cases when there is only one possible way to link two classes (and most of the cases are like this) we want to give user a simple syntax like 'A ->B' in DOT. We don't want the user to specify redundant information in the script. As we are anyway handling graph validation in the FactoryFacade class it seems to be a reasonable place to resolve reference names.

3.2.3 Implementation

The implementation of the Factory API was very straightforward. We created a class FactoryFacade according to the design described above.

However, there is one tricky moment related to the string bounds. As we mentioned in Section 3.1 a string bound can be either global or for particular attribute. When the user sets a value for a string attribute to a particular string, there can be two variants. If the length of the string is less then the current bound, the bound remains the same, otherwise the bound is set to the length of the string.

When the ASMIG string generator generates a string it allocates exactly bound number of characters. It means that if the bound is two, every value of this attribute will have length two. The same applies to predefined values as well. If the bound of the attribute is five, and the value is three characters long, ASMIG will generate values for remaining two characters.

Let us consider the following code:

```
FactoryFacade ff = new FactoryFacade("model.ecore");
addNode("A", "a1")
addNode("A", "a2")
setAttribute("a1", "name", "Alice");
setAttribute("a2", "name", "Bob");
```

After the first `setAttribute()` call the bound on attribute name will be equal to 5. After the second call it does not change. In the instance the name of node a1 will be 'Alice', but the name of a2 will be something like 'Bobaa' because ASMIG will generate the missing characters so that the resulting length will be equal to bound.

To prevent this behaviour we add a zero character (\0) to every string value. It does not affect ASMIG generation process, but when the string is interpreted it finishes on a zero character.

3.2.4 Testing

As one of our main priorities is code reliability, testing plays an extremely important role in the development process.

In this first iteration we are only testing the FactoryFacade class with unit-tests. We are using black-box Equivalence Partition testing as described in [32] with the elements of Combinational testing (when we have to invoke a series of methods to test some case). Finally we perform white-box testing to achieve good code coverage.

The test cases in detail are shown in Appendix A. We instantiate a FactoryFacade object, then call one or several methods that are supposed to create some graph. To check expected values we get a factory in the following way:

```
Factory f = factoryfacade.getE2Atranslator().getFactory();
```

The designed test cases were implemented in 23 unit tests. There are less unit tests then test cases because non-error test cases can appear in one unit test. For example, cases 5 and 7 from Appendix A are tested in one test.

This test set gives us a coverage of 91.9% (measured by EclEmma[2]).

To achieve full statement coverage we added 5 white-box tests. The coverage we got is 98.9% which corresponds to full statement coverage and several missed branch which is a sufficient level of coverage for our case.

To summarize, we performed unit-testing of Factory API. In figures: 28 tests, 98.9% coverage, all tests pass.

As the result of testing several minor bugs were fixed. We would like to mention one of them. The API didn't throw the exception on the attempt to set a value for not existing attribute (test case 10). This is not a critical bug, but this error could have caused problems at a later stage when the user expects the attribute value to be set, but it is not because of spelling mistake, and there is no signal.

3.3 Iteration 2: Language Metamodel

3.3.1 Design

In this iteration we are starting to use the Eclipse Modelling Framework. The main artifact of the iteration is a language metamodel in Ecore format. Additionally, we are creating the integration mechanism to convert a graph from ASMIG format into the DSL format.

The structure of the incremental part of this iteration is shown in Figure 3.8.

The metamodel is shown in Figure 3.9. This is in a terms of domain-specific languages a semantic model of a language. This is a data structure that will store the DSL program after parsing. Our language is for defining attributed graphs, and this is reflected in the metamodel.

The root of the metamodel is a 'Graph' classifier. The Graph has an ecoreFilePath attribute which stores the path to the Ecore model for a current DSL script. The Graph contains a collection of Nodes and Edges. The type of a Node is a name of an EClass from the model. An Edge has a referenceName attribute which is optional - as we explained before in many cases this information is redundant. Finally, a Node contains a collection of Attributes. EMF supports two types of references - Containment (drawn with the black diamond) and Non-Containment (drawn as simple arrows).

[2]`www.eclemma.org`

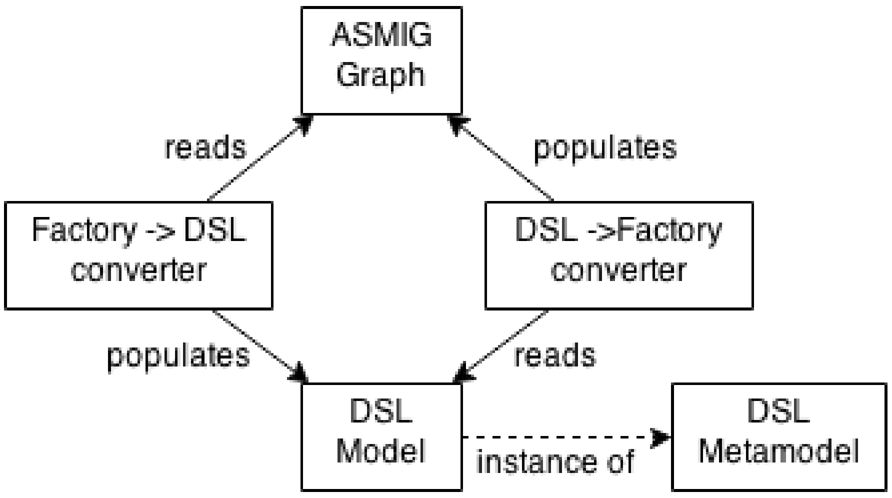


Figure 3.8: Iteration 2 design. DSL Metamodel is integrated with ASMIG graph representation

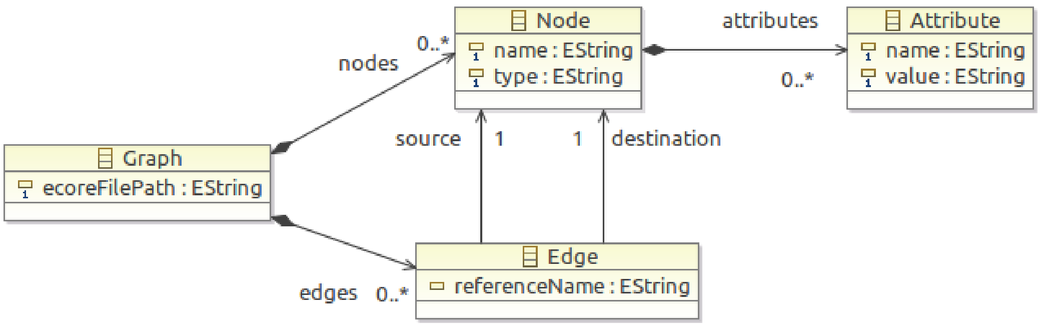


Figure 3.9: Metamodel of the DSL for defining the instance template graph

Apart from the DSL metamodel we have a DSL model which is an instance of a metamodel and represents a concrete graph. A model is generally a result of parsing a script, but here we will instantiate it programmatically in our Factory-DSL converter.

3.3.2 Implementation

As we are dealing with EMF framework our implementation in many asbects is based on the standard techniques for EMF described in [33].

There are three ways to create a model in EMF:

1. Create from scratch in the editor.

2. Import from UML.

3. Generate from annotated Java code.

For our case we picked the first option. First of all, the model is not large. And secondly, the current ASMIG Java code is not that straightforward to annotate several classes and get a model like ours (compare Figures 3.9 and 3.3).

One of the benefits of using EMF is that from a model it generates first of all a 'DSL factory' that provides methods for creating instances of this model, and also several utility classes one of which is a 'DSLSwitch' class for traversing a model instance.

After creating a metamodel we had to create the converters between the ASMIG format of a graph and a graph stored as an instance of DSL metamodel.

For converting ASMIG to DSL we took a following approach. The ASMIG factory class has `toString()` method used for printing the graph for debugging purposes. This method walks the BVG, ENA and ST collections, gathers the necessary information and forms a string.

We took the logic of walking the graph and created our own method where we replaced the string formatting with calls to DSL factory API.

For the opposite conversion from DSL to ASMIG we used the instance visitor class generated by EMF (see figure 3.10).

The classes Switch and DSLSwitch are generated by EMF. Class DSLFactoryVisitor implements the visiting functionality for different parts of the instance and uses the FactoryFacade to create a graph in ASMIG. A DSLFactoryVisitor overrides methods of DSLSwitch by making appropriate FactoryFacade calls while visiting the DSL graph.

Here is the example of usage of this class:

```
DSLFactoryVisitor visitor = new DSLFactoryVisitor();

for(
    Iterator<Object> itr = modelInstance.getAllContents();
    itr.hasNext(); )
{
    EObject eObject = (EObject)itr.next();
    visitor.doSwitch(eObject);
}

return visitor.getE2Atranslator();
```

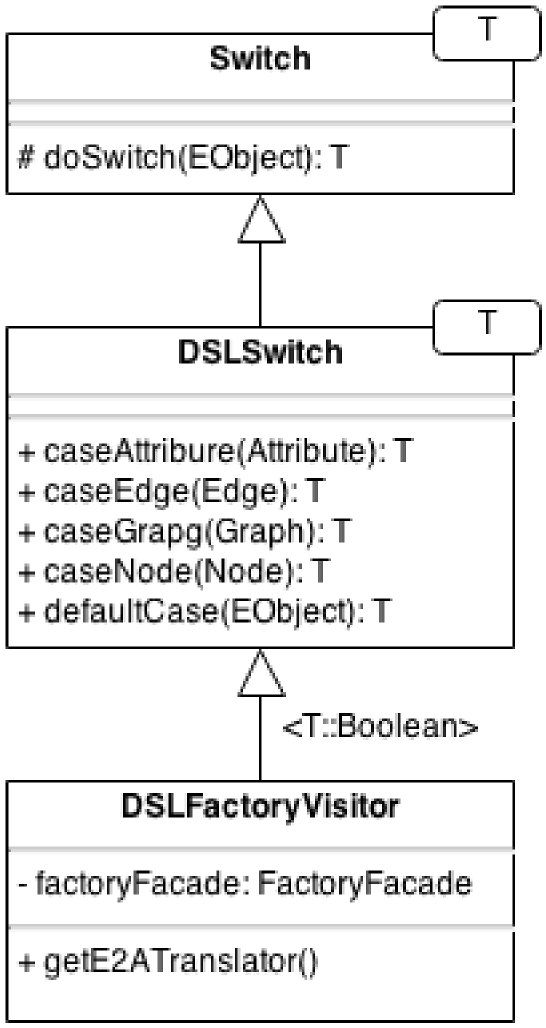


Figure 3.10: DSL Visitor class hierarchy. Switch and DSLSwitch classes are generated by EMF, DSLFactoryVisitor traverses the DSL Model and populates the Factory via the API in FactoryFacade

There is a return value T which is not used in our case. We just return Boolean.TRUE in every overriden method.

3.3.3 Testing

To test the new functionality of graph transformation we perform integration testing rather than unit testing.

The approach of use is illustrated in Figure 3.11.

Here the ASMIG graph is instantiated, then converted into DSL graph,

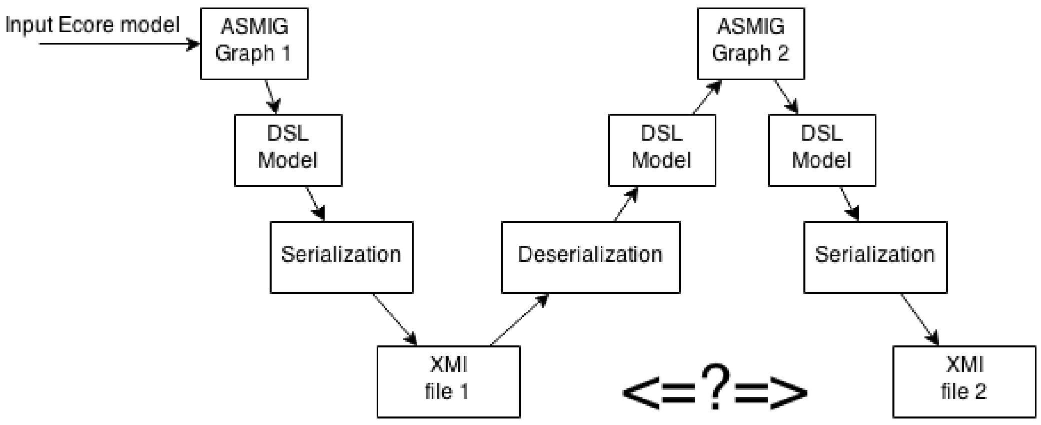


Figure 3.11: Integration testing schema. We produce two XML files and compare them

then serialized in XMI file. This file is then deserialized, converted back to ASMIG and then again to DSL and serialized as XMI.

XMI (XML Metadata Interchange) is a default format for serializing model instances in EMF and the serialization/deserialization functionality is available out of the box.

We chose this approach because it is easier to compare two XML files than two EMF models. However, we have to admit this approach did not work. The files contained equivalent models, but the order of nodes and edges was different. This happens because of different way we create ASMIG graph - the first time it is created using the ASMIG API, and the second time we create every node separately using our FactoryFacade API.

In spite of this we decided to compare the DSL models by comparing the files. EMF has a built-in comparison mechanism for comparing models. However, it also takes order into consideration and therefore treats similar models as different.

This issue of ignoring the order of elements while comparing EMF instances is not new. There is a post in Jorge Manrubia's blog addressing this subject[3].

The solution is to create a subclass of `EcoreUtil.EqualityHelper` class which performs the comparison and override the comparing of the lists in following way: first we sort both lists, then we compare sorted lists.

As this approach involves interfering with EMF inner structure we decide to test it first on sample models. Our custom comparator takes two file

[3]`http://jorgemanrubia.net/2008/07/06/comparing-emf-models/`

names as input and produces a boolean result of comparison.

We again used Equivalence Partitions technique to test the component.

Method under testing:

```
boolean compareXMIFiles(String fileName1, String fileName2);
```

EP for fileName1
1*) not existing file
2*) invalid XMI file
3) valid file

EP for fileName2
4*) not existing file
5*) invalid XMI file
6) valid file

EP for output
7) true for identical files
8) true for files with different order of nodes
9) true for files with different order of edges
10) false for files with difference in node attribute
11) false for files with difference in node
12) false for files with same nodes and difference in edge

This test set is implemented in 9 unit tests and covers 100% of the method under testing.

After the testing of the comparator we performed the integration testing as explained above. In the process of testing several defects has been discovered. First of all, we found a bug in FactoryFacade class related to handling a model with multiple references having the same name. Secondly, there was an error with handling bidirectional references.

After the integration testing we also did system testing using the Z3 prover to make sure that after the ASMIG-DSL-ASMIG conversion loop the generated instances are the same. This is described in detail in the Evaluation chapter in Section 4.1.

3.4 Iteration 3: Language syntax

3.4.1 Design

In this iteration we will design the syntax of the language and bind it to the previously created metamodel as shown in Figure 3.12.

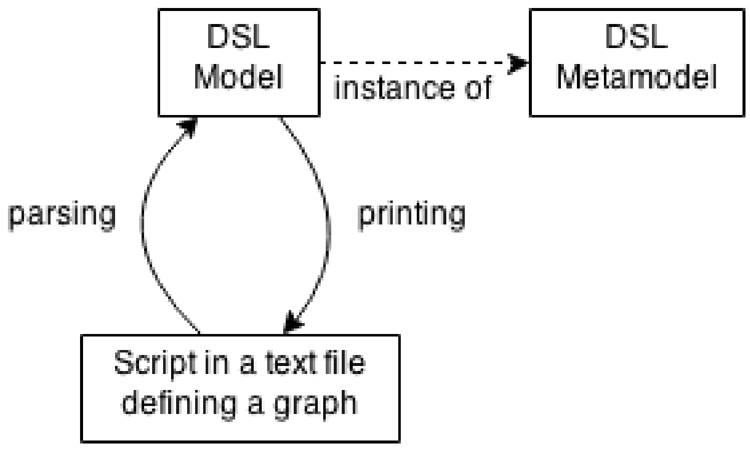


Figure 3.12: Iteration 3 components: binding of the textual language to the metamodel

Our language should be able to express the following concepts:

- Declaring a node with its type.
- Specifying the values of node attributes.
- Declaring an edge between two nodes (optionally with the reference name to avoid ambiguity).
- Specifying the corresponding Ecore model.

Our syntax design was mainly inspired by Graphviz DOT language because first of all DOT is quite similar to what we need and secondly it is a well-known language and many people are already familiar with its syntax.

The grammar of our ITG language in the Backus-Naur Form is shown in Figure 3.13.

The root element `<itg>` of the script consists of a reference to the Ecore file followed by the definitions of nodes and edges.

For the node definition we used a slightly modified syntax of node definition in the DOT language. The difference between the nodes in our language and in DOT is that in our language the node has a type. So the node declaration in our language looks like this:

```
Class c1 [isAbstract=true, id=25]
```

The attributes block is optional.

As for the edges, we decided to support two types of syntax. The first type of edge definition is the same as in the DOT language, with an optional reference name in square brackets:

```
<itg> ::= <ecore-reference> <graph>
<ecore-reference> ::= "model" <quoted-ecore-file-path>
<quoted-ecore-file-path> ::= '"' <ecore-file-path> '"'
<graph> ::= <node-declarations> <edge-declarations>
<node-declarations> ::= <node> | <node> <node-declarations>
<node> ::= <node-type> <node-name> <optional-attributes>
<optional-attributes> ::= "" | "[" <attributes-list> "]"
<attributes-list> ::=
    <attribute> | <attribute> "," <attributes-list>
<attribute> ::= <attribute-name> "=" <attribute-value>
<edge-declarations> ::= <edge> | <edge> <edge-declarations>
<edge> ::= <edge-arrow> | <edge-dot>
<edge-arrow> ::=
    <source-node-name> "->" <destination-node-name>
    <optional-reference-name>
<optional-reference-name> ::= "" | "[" <reference-name> "]"
<edge-dot> ::=
    <source-node-name> "." <reference-name>
    "=" <destination-node-name>
```

Figure 3.13: Language grammar

```
class1 -> operation1 [ownedOperation]
```

The second variant of edge syntax is taken from the usual Java-like syntax for qualified names which refers to a property of an object using a dot:

```
class1.ownedOperation = operation1
```

We had following reasons for including this type of syntax. First of all, for some models it may be more illustrative to use this syntax instead of the 'arrow' syntax. Secondly, we believe supporting this standard notation makes it easier to integrate other systems with ASMIG+. Finally, it is an easily understandable syntax used in many other languages which is clear even without knowing anything about our language.

3.4.2 Implementation

The implementation of the language is done using the EMFText language workbench.

The language development process in EMFText consists of the following stages:

1. Create a language metamodel in Ecore format.

2. Define grammar rules that specify how the script should be translated into an instance of the metamodel
3. Generate the tools - scanner, parser and other utilities.
4. Optionally customize the generated code.

As we already have our metamodel, we start with the grammar. In the EMFText the grammar is stored in a 'Concrete Syntax' file. This file stores the rules for every metamodel EClass in a form which is close to Backus-Naur notation.

Here is a fragment of the file that corresponds to the grammar defined in Figure 3.13.

```
START Graph
RULES {
  Graph ::=
    "model" #1 ecoreFilePath['"','"'] !0 !0 nodes* !0 edges*;

  Node ::=
    type[] name[]
    ("[" !1 attributes ("," !1 attributes)* !0 "]")? !0;

  Attribute ::=
    name[] "=" value['"','"'];

  Edge ::=
    (source[] #0 "." #0 referenceName[] "=" destination[] !0)|
    (source[] "->" destination[] ("[" referenceName[] "]")?);
}
```

The statement `START Graph` means that the root node of the syntax tree will be a Graph object. The `RULES` section contains the rules for every EClass of our metamodel (see Figure 3.9). The rule for every class tells the parser what syntactic construct to expect when the object of the class is expected. The attributes of classes are referred to with the square brackets (e.g. `name[]`, `value[]`). The construct `ecoreFilePath['"','"']` means that the value of ecoreFilePath attribute will be written between double quotation marks.

The EReference syntax differs for Containment and Non-Containment references.

Non-Containment references are defined like attributes (source and destination references in the Edge class). To identify the object which is referred to by a Non-Containment reference EMFText is looking for the attribute ‘id’ or attribute ‘name’. Since the Node class has the attribute ‘name’, it will be used as a unique identifier for Node objects.

Containment references are written without square brackets and when a Containment reference is encountered, the parser expects an expression defining a contained object. For example, for a Node object the Attribute objects (which are referenced with a Containment reference) will be defined inside the Node object according to the rule for Attributes (name = value), but for Edge references (source and destination) only the Node name will be expected.

Additionally, EMFText supports the optional parts in the rules (?) and multiple occurrences (*).

The expressions like ‘#1’ and ‘!0’ control the formatting of the script when the model is printed into a text file. The expression ‘#n’ prints n whitespace characters, and ‘!n’ means print a line break followed by n tabs.

When there are several alternative ways to represent the object (for example, the Edge), the printer uses the first option.

Having specified this Concrete Syntax file, we can generate the parser and the printer. The parser is used to transform the text file containing the script into the model containing the corresponding objects. The printer performs the opposite task. As our language is very simple, we don’t need any customization of the generated code. Therefore, we have both our ‘parsing’ and ‘printing’ arrows from Figure 3.12 done.

An example of how the DSL code looks like for a model describing BibTeX format is shown in Appendix B.

3.4.3 Testing

We perform the testing of the third iteration in a similar way to the testing of the second iteration (see Subsection 3.3.3).

First we perform Integration Testing to define the consistency of the conversion from the script to the model and back. This testing seems to be a bit redundant because the conversion is performed by the EMFText-generated code. However, since we already have the code for checking the equivalence of two instances of our DSL Metamodel, we decided to test the third iteration as well.

The test scenario for the integration testing is following:

1. Load Ecore model to ASMIG, generate ITG
2. Convert ITG into DSL representation (DSL model 1)

3. Print DSL model into text file in a form of script
4. Parse script, get DSL model 2
5. Compare DSL model 1 with DSL model 2

System testing has also been performed. For the system testing we compared the ASMIG system with the Ecore model as the input and the ASMIG+ system with the equivalent DSL script as the input. The testing strategy is described in Section 4.1.

As a result of testing no new errors have been found in this iteration. Presumably we caught the majority of the defects in the earlier stages of testing.

Chapter 4

Evaluation

The evaluation of our work includes answering two questions. The first question is ‘Does the system do what it is expected to do?’. This question is related to the system testing and it is covered in the corresponding section. The second question is ‘Does the system help the user to achieve their goals?’. This is where the true evaluation comes in. In other words, is our new system applicable for generating test data sets for given models? We will address this question in the ‘Experiment’ section of this chapter where we will use the system to generate test data for our DSL.

4.1 System testing

To perform system testing of ASMIG+ we need to check that for the input model in the form of a DSL script the set of generated instances will have the following properties:

1. Every generated instance is an instance of the input model.
2. For the bounded search space the generated set contains all possible instances of the model.

Luckily we have the ASMIG system at our disposal which we assume is already tested. Therefore, we can reduce the task of system testing ASMIG+ to the task of checking whether ASMIG and ASMIG+ give equivalent outputs for equivalent inputs.

To get equivalent inputs for ASMIG and ASMIG+ we will use our ASMIG-to-DSL conversion mechanism which we already tested in the phase of integration testing.

However, the task of defining output instance equivalence is itself rather complicated. First of all, it is not feasible to generate a complete set of instances except for trivial models.

Secondly, the order of generated instances may differ for ASMIG and ASMIG+. This means the first N instances produced by ASMIG may be different from the first N instances produced by ASMIG+, although all of these instances might be correct.

In spite of these facts we decided to further reduce the task of system testing. We know that the instance generation itself is performed by Z3. After an instance template graph is translated to a boolean formula there is no difference in the processing of this formula by Z3 for ASMIG and ASMIG+. So rather than comparing the output of Z3 (instances) we will be comparing the input to Z3 (boolean formulas). A good point of this approach is that the equivalence of boolean formulas can be formally proved.

4.1.1 Verifying the equivalence of graph formulas with Z3

The interaction of ASMIG with Z3 is performed in following way. The ITG is translated to a formula which is then exported to a file in SMT-LIB v2 format [34] for processing by Z3.

So our task here is easy - take Z3 file from an ASMIG run and compare it to the file from an ASMIG+ run. However, we encountered an unexpected problem. The variables in the file denoting edges were named differently in different runs. The edges were named sequentially e1, e2, e3 etc. throughout the whole model. There was an edge e1 in file a.z3 and an edge e1 in a file b.z3 and these edges were between different nodes.

To overcome this problem we had to change the naming of edges. As we stated earlier, one of our principles was not to change ASMIG code. However, in this case it was absolutely necessary. The danger of this modification is that it may result in errors if some logic in another place of the system was depending on the old edge naming rule.

We decided to guard ourselves with a smoke test. This test iterates through a set of test models and generates one instance for every model with ASMIG. After this we make the naming change and then we run the smoke test once again to make sure nothing has broken. The details and the results of this smoke testing are in Subsection 4.1.2.

The new naming convention for edges we used forms the edge name in the following way:

```
    edgeName =
        "e_"+referenceName+"_"+sourceNodeName+"_"+destNodeName;
```

Having made this change we can come back to the system testing. We have two Z3 files - one from an ASMIG run and one from an ASMIG+ run on the equivalent input. These files are not identical, but they are supposed to describe equivalent formulas.

The structure of generated Z3 files is shown in Figure 4.1. Every file starts with a header which is always the same. Then a number of variable declarations follow. These variables describe the presence of nodes and edges in the instance and the values of node attributes. Then the formula itself is encoded in a number of asserts. At the end of a file there is a `check-sat` statement which checks if a model is satisfiable or not and a `get-value` statement which retrieves the combination of values for variables that makes all asserts true.

```
(set-option :print-success false)
(set-option :produce-models true)
(set-logic QF_LIA)

(declare-const class1 (Bool))
(declare-const class2 (Bool))
(declare-const operation2 (Bool))
(declare-const property1 (Bool))
(declare-const e_ownedOperation_class1_operation1 (Bool))
(declare-const e_parents_class2_class1 (Bool))
(declare-const class1_hash (Int))
(declare-const class2_isAbstract (Bool))
...

(assert (=> (= class1 false) (= class1_hash (- 1))))
(assert (=> (= class1 false) (= class1_isAbstract false)))
(assert (=> (not class1) (= class1_class1_name_0 0)))
(assert (=> (not class1) (= class1_class1_name_1 0)))
...

(check-sat)
(get-value (...))
```

Figure 4.1: Structure of Z3 file

To compare two models from two files of this structure we merge them into one Z3 file using the following algorithm. First we read the `declare-const` statements into two lists, sort the lists and compare the sorted lists element-by-element. If the lists are equal, we write these variable declarations to the file.

Then we change every `assert` statement into a boolean function taking no parameters. In this way we turn asserts into functions A1, A2, A3 etc. for the first file and B1, B2, B3,... for the second file.

Then we define functions A and B:

$$A = A_1 \wedge A_2 \wedge A_3 \wedge ...$$

$$B = B_1 \wedge B_2 \wedge B_3 \wedge ...$$

Function A defines the model from the first file, B corresponds to a model in the second file. These boolean functions are equivalent if and only if A implies B and B implies A. Therefore, the expression $(A \implies B) \wedge (B \implies A)$ should be a tautology. Z3 cannot tell if a function is a tautology. Instead, it can tell whether a function is satisfiable or not. In other words, we can check for a contradiction. We define a function which is supposed to be a contradiction:

$$NotEquivalent = \neg((A \implies B) \wedge (B \implies A))$$

Finally, we assert $NotEquivalent$. If the models A and B are equivalent, $NotEquivalent$ is a contradiction and consequently Z3 will give an answer that the model is unsatisfiable.

We then feed this merged file into Z3 and if the output is 'unsatisfiable' then the inputs to Z3 from ASMIG and ASMIG+ are equivalent and it means that our system test passed.

4.1.2 Smoke testing

The purpose of smoke testing is to find out whether the system works 'in general' or not. This term originates from electrical engineering when a device could actually smoke when turned on the first time. In software testing this kind of testing checks that the software does not crash. It is usually a kind of testing performed regularly after system changes to make sure other parts of the system were not affected.

In our case we performed smoke testing after changing edge naming in ASMIG. Of course it does not guarantee the system did not break, but it gives at least some confidence. Our test scenario was the following:

1. Generate one instance for every model from a test set and save these instances (*.dot files).

2. Make a change in the code.

3. Generate the instances for every model once more.

4. Compare the size of new and old instances, if the difference is more than 10% of file size then fail the test, else pass.

We set this threshold of ten percent because instances are often different even between two runs without code changing. However, if the files differ significantly it is suspicious.

The smoke testing revealed one error. The edge name after modification includes the name of the reference. In one of the test models a reference name contains a colon symbol (':'). This colon got inside the edge name and caused a crash of Z3 processing as Z3 does not allow identifiers containing a colon.

Our solution was to replace all occurrences of ':' with '_colon_'. Of course if there are two references in the model named 'a:b' and 'a_colon_b' we will still get an error, but first of all this is unlikely and secondly this is inevitable because for reference names we can use more symbols then for Z3 identifiers.

4.1.3 Set of test models

The ASMIG project includes a data set of sample Ecore models for testing. This set is supposed to cover all features of Ecore which somehow participate in the instance generation process.

The test set includes 84 models either taken from the Metamodel Zoo[1] or created manually. The smallest models consist of one class. The largest models have more than two hundred classes. The models we used for the testing are mainly metamodels of some languages - Java metamodel, UML metamodel, ECore metamodel itself etc.

Ten largest models from the set are shown in Table 4.1.

The model java.ecore has the biggest number of classes - 233. However, the largest model in the set is UML2.ecore which has 227 classes, 437 references and 91 attributes. Although it has a slightly smaller number of classes, it has the biggest number of references which increases processing time dramatically.

We also added our language metamodel dsl.ecore (shown in Figure 3.9) to the test set so in total we got 85 models to test our system on.

The information on the remaining 75 models can be found in Appendix C.

[1]`http://www.emn.fr/z-info/atlanmod/index.php/Ecore`

Model	Classes	References	Attributes
java.ecore	233	104	17
UML2.ecore	227	437	91
XHTML.ecore	146	255	33
JavaAbstractSyntax.ecore	95	151	46
KML.ecore	93	2	64
ATL.ecore	84	114	32
ACG.ecore	70	40	20
HTML.ecore	59	14	98
MavenMaven.ecore	58	34	94
Ant.ecore	48	28	93

Table 4.1: The largest models from the test set

4.1.4 Test results summary

In this subsection we would like to summarize how we tested different components of the system and give some results.

The components covered by unit-tests:

- FactoryFacade class: 28 unit tests that give 98.9% of code coverage.
- XMIComparer class: 9 unit tests, 100% of code coverage.

Integration testing. Performed by round-trip conversion of models between ASMIG Factory format and DSL format:

- Integration between DSL Metamodel and ASMIG Factory. All 85 test models pass. Total testing time: 2.5 minutes.
- Integration between DSL Scripts and ASMIG Factory. All tests pass. Total testing time: 23.6 minutes.

System testing. Performed by verifying the equivalence of Z3 inputs:

- On the second iteration (conversion to DSL model). All 85 test models pass. Testing time: 29 minutes. The long operations are translation from ITG to a boolean formula which is performed twice (for ASMIG and ASMIG+) and processing the merged Z3 file (for the largest model file is about 120MB).
- On the third iteration (conversion to DSL script). All tests pass. Total testing time: 49 minutes.

Smoke testing. Performed for detecting system crashes after code modifications. Testing time for the test set of 85 models: 10 minutes.

Additionally, we did performance testing to find out the overhead of converting the ITG into the corresponding script and back. On small models the overhead is not significant (see Figure 4.2) as the most time-consuming part is reading the input Ecore model. Stages performed by our code are shown in red.

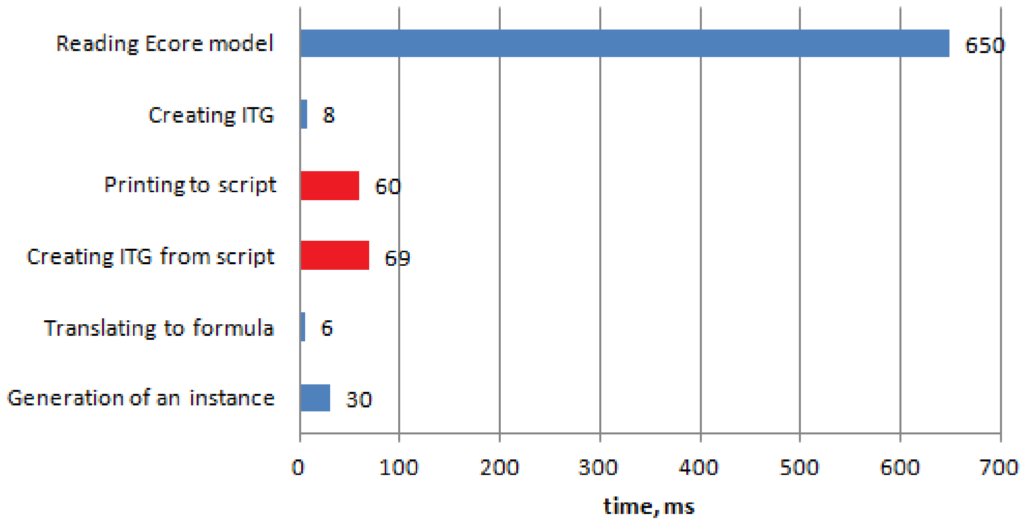


Figure 4.2: Processing time in milliseconds for a small model ‘dsl.ecore’. The longest operation is reading the model. Overhead is insignificant

However, on the large models the situation changes dramatically. Figure 4.2 shows the processing time distribution for the largest model UML2.ecore.

We can see that the printing of the ITG to script file takes the longest time. This stage consists of conversion from the ASMIG ITG to the DSL model (which takes less than a second) followed by the serialization of the model to a file which is performed by the EMFText and takes almost 20 minutes for UML2.ecore.

The bottle-neck of the ASMIG+ system is clear. The next stage would be the investigation of the EMFText serialization mechanism and trying to optimize it to reduce total processing time.

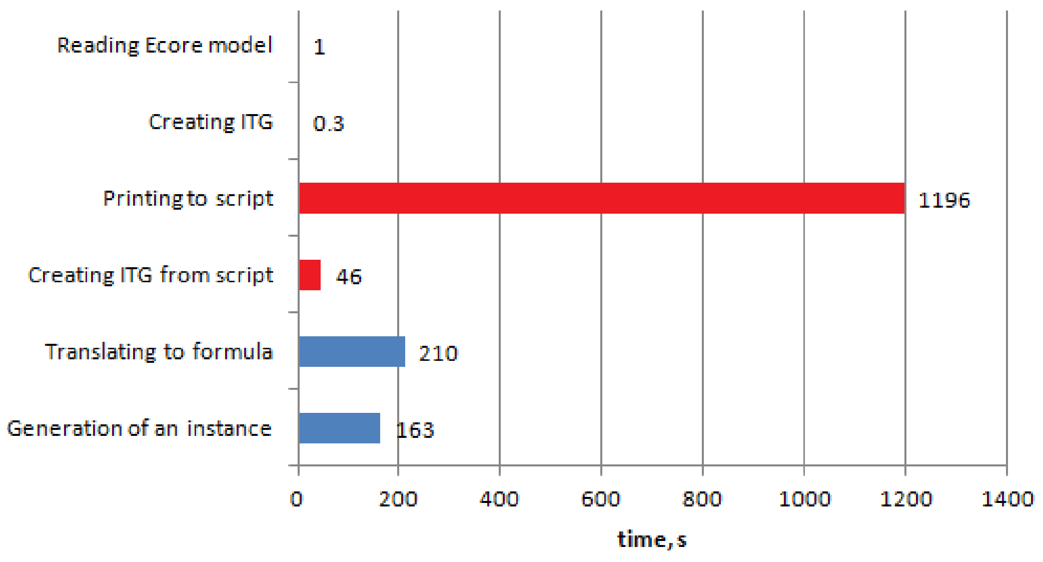


Figure 4.3: Processing time in seconds for the model 'UML2.ecore'. The export of a graph to the textual representation performed by EMFText takes 74% of the whole processing time

4.2 Experiment

In this section we would like to show how the ASMIG+ system can be used to generate test data. We will be testing the ASMIG+ system itself. We have a metamodel of our DSL (see Figure 3.9) and we want to generate a set of scripts in this DSL in order to run our system with these scripts as the input.

Let us have a look at the instance of DSL metamodel generated with the ASMIG system. Figure 4.4 shows one of the generated instances.

As we can see, this instance conforms to the metamodel. It has one Graph with two Nodes, two Edges and two Attributes, but this instance is semantically invalid.

First of all, the `ecoreFilePath` attribute in the `graph1` object does not contain a link to an Ecore model. Secondly, the Nodes don't have unique names. Finally, both edges have `node2` object as source and destination. It can be fine as long as the edges stand for different references. However, in this instance both edges have the same reference name 'aa'.

It is clear that this instance cannot be used to produce a valid DSL script. However, we can use the new ASMIG+ functionality to customize the instance template so that the generated instances would be valid.

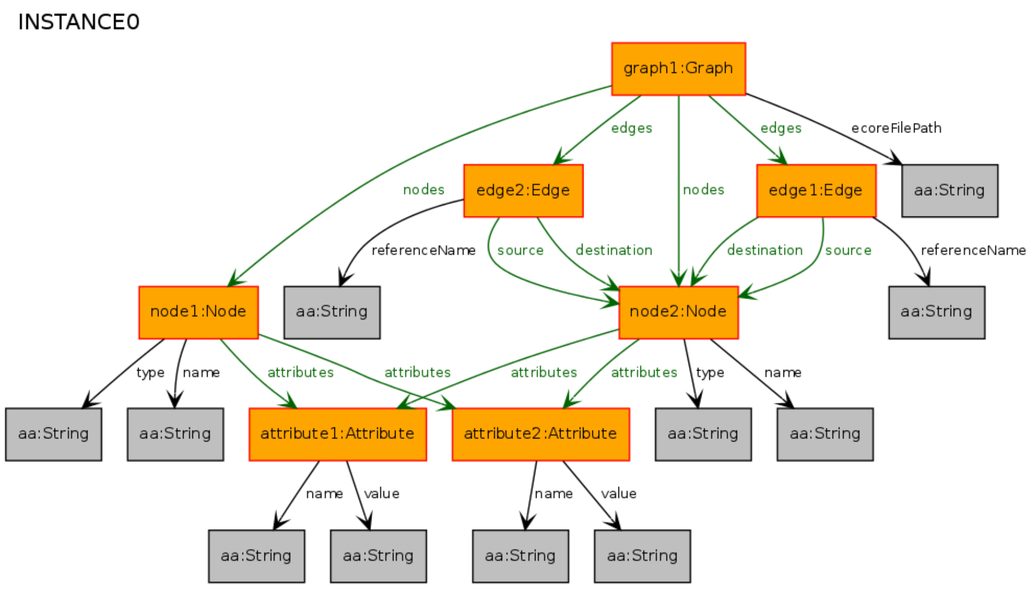


Figure 4.4: A generated instance for the model dsl.ecore without template customization

The instances of the DSL metamodel are template graphs. So we have to customize the template for templates. For the sake of clarity we will call this template a *metatemplate*.

Figure 4.5 illustrates the test scenario.

We decided to generate templates for the model Sample.ecore shown in Figure 2.6 because it is simple and has all the elements that an Ecore model typically has.

First of all we export the ITG generated by the ASMIG system into a script. Here is this script containing our metatemplate:

```
model "dsl.ecore"

Edge edge1
Edge edge2
Graph graph1
Attribute attribute1
Attribute attribute2
Node node2
Node node1

node2.attributes=attribute2
```

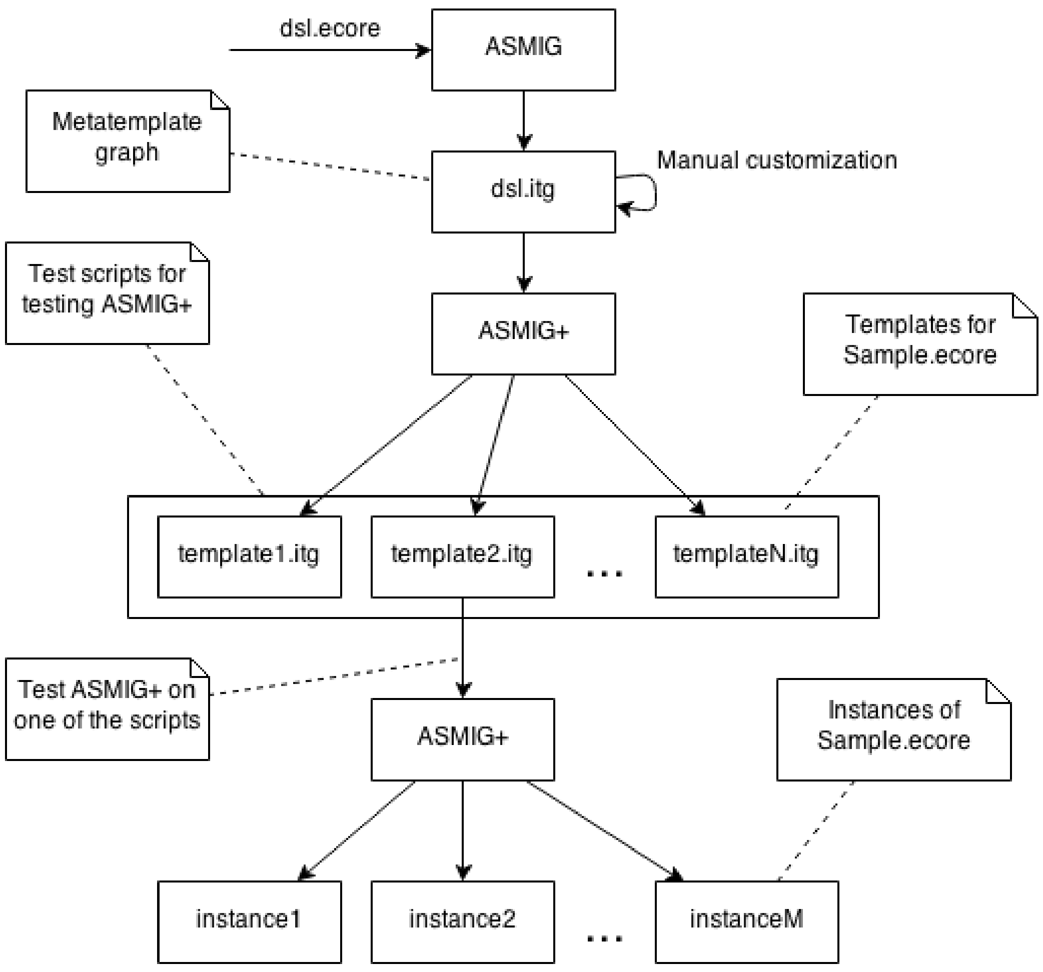


Figure 4.5: Test scenario for the templates generation

```
node2.attributes=attribute1
node1.attributes=attribute2
node1.attributes=attribute1
edge1.source=node1
edge1.source=node2
edge2.source=node1
edge2.source=node2
edge1.destination=node1
edge1.destination=node2
edge2.destination=node1
edge2.destination=node2
```

```
graph1.nodes=node1
graph1.nodes=node2
graph1.edges=edge2
graph1.edges=edge1
```

We have to set the ecoreFilePath in the graph1 object to an Ecore model:

```
Graph graph1 [ecoreFilePath = "Sample.ecore"]
```

Every Node in the metatemplate represents a possible node in the template. So the types of the nodes should be EClass names from Sample.ecore model. Edges have to link nodes according to the EReferences of Sample.ecore.

Here is the edited version of the metatemplate script:

```
model "dsl.ecore"

Graph graph1 [ecoreFilePath = "Sample.ecore"]
Node class1 [type = "Class", name = "c1"]
Node class2 [type = "Class", name = "c2"]
Node prop1 [type = "Property", name = "p1"]
Node op1 [type = "Operation", name = "o1"]
Edge parents [referenceName = "parents"]
Edge property [referenceName = "ownedAttribute"]
Edge operation [referenceName = "ownedOperation"]

parents.source=class1
parents.destination=class2
property.source = class1
property.destination = prop1
operation.source = class2
operation.destination = op1

graph1 -> class1
graph1 -> class2
graph1 -> prop1
graph1 -> op1
graph1 -> parents
graph1 -> property
graph1 -> operation
```

When we use this edited ITG file (metatemplate) as the input to our ASMIG+ system, it produces a set of instances for this metatemplate (templates). Some of the generated templates are shown in Figures 4.6, 4.7 and 4.8. The orange boxes represent the objects and the grey ones represent the attributes.

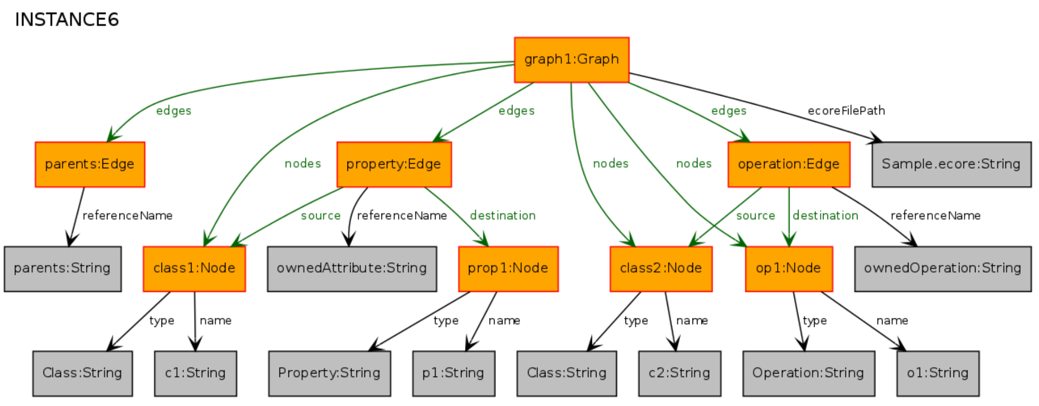


Figure 4.6: Generated template 6. The Edge ‘parents’ does not have source and destination nodes, The Node ‘prop1’ is not connected to the Graph. Therefore, the Edge ‘property’ is also meaningless

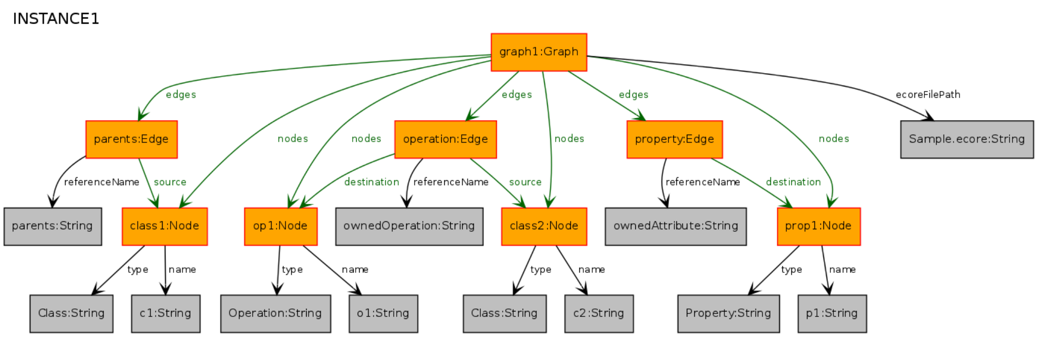


Figure 4.7: Generated template 1. The ‘parents’ Edge has a source, but does not have a destination

These instances correspond to the block ‘Test scripts for testing ASMIG+’ in Figure 4.5. As can be seen, there are meaningless parts in the generated templates. For example, in Figure 4.6 the Edge `parents` does not have source and the destination nodes. In Figure 4.8 the Edge `parents` does not belong to the graph, so it will not be present in the instances.

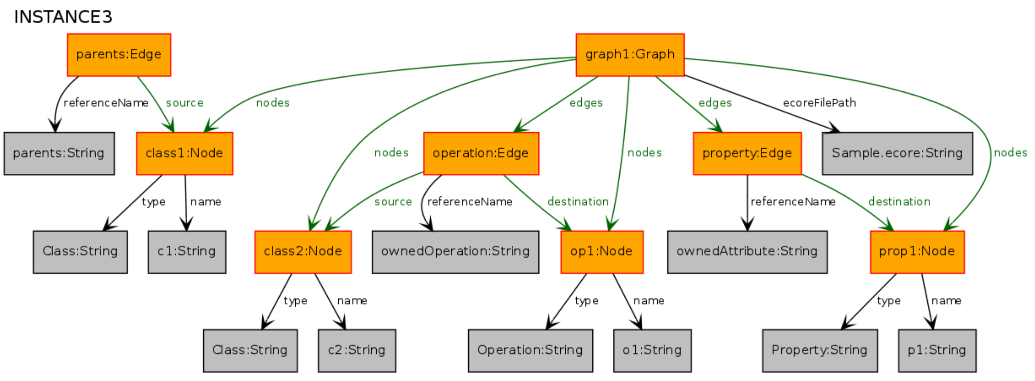


Figure 4.8: Generated template 3. The ‘parents’ edge is not connected to the Graph, ‘property’ edge has no source

Here we are coming to one of the limitations of this approach of defining the instance template graph - in the current implementation of ASMIG we cannot force some elements from the template graph to be present in every instance.

The next step is to convert the generated templates to the ITG scripts. We did it manually for the instance in Figure 4.6. However, it is easy to implement an automatic converter. The resulting script is shown in Figure 4.9.

Notice that the node `prop1` is ignored because there is no link between the node `graph1` and the node `prop1`. Similarly, edges `parents` and `property` are not present in the script.

After obtaining this script we perform the final part of our experiment - actually running the ASMIG+ system on this generated test input.

The instance generation for the input in Figure 4.9 has been successful and we got a number of instances of Sample.ecore model. See Figures 4.10 and 4.11.

Generation of these instances in Figures 4.10 and 4.11 conclude our experiment according to the Figure 4.5

To summarize, we tested the processing of our DSL by the ASMIG+ system, using the generated programs as the input. The test programs have been generated by the ASMIG+ system itself with the DSL metamodel as the input. The applicability and the limitations of this approach will be discussed in the next chapter.

```
model "Sample.ecore"

//the object class1 from Figure \ref{fig:itg6}
//represented by with type and name attributes
Class c1

//the object class2 from Figure \ref{fig:itg6}
Class c2

//the object o1 from Figure \ref{fig:itg6}
Operation o1

//this edge corresponds to the operation object
//which has class2 as a source and op1 as destination
c2 -> o1
```

Figure 4.9: ITG Script for template 6 created manually

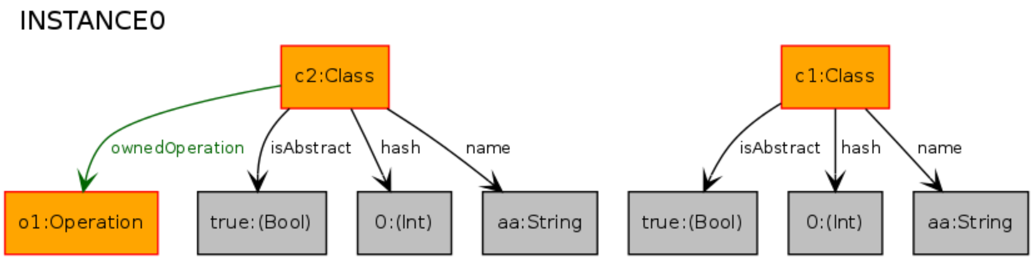


Figure 4.10: Generated instance 0 for a template graph in Figure 4.9

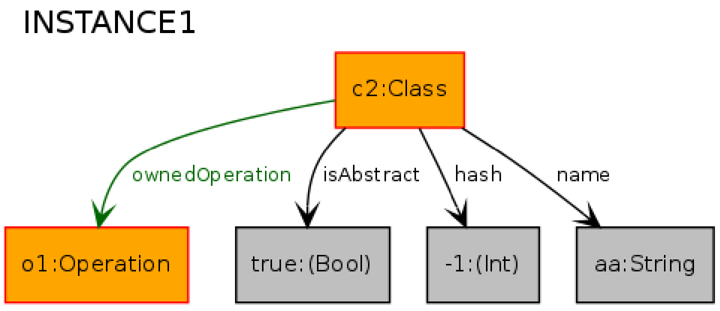


Figure 4.11: Generated instance 1 for a template graph

Chapter 5

Conclusions and future work

In this chapter we would like to summarize our work, draw some conclusions and discuss potential future work.

Our aim was to make the model instantiation in ASMIG more usable for test data generation. We fulfilled this aim by providing a language for defining the instance template graph and the functionality to use a script in this language as the input to the system.

Now let us discuss what we can and what we cannot do with our ASMIG+ system. The system performs an automatic generation of model instances. The input to the system can be either an Ecore model or a script defining a template graph for instance generation. We can export the template graph generated by ASMIG for an input Ecore model into a script. Therefore, we can interfere into the generation process which makes it more transparent. We can edit the template script: first of all we can add or remove nodes and edges to achieve the necessary configuration of the template. Secondly we can set values of node attributes, and these values will then appear in the generated instances. This setting of the attribute values is very important because with this feature we can generate not just random instances, but the instances that will be semantically valid for the system under test. However, there are a number of limitations.

The main limitation in our opinion is that there is no way to ensure that certain elements of the template appear in the instance. It is always decided by Z3 which nodes and edges are switched on and off in the instance. In Section 4.2 we faced this limitation when the generated instances had some edges without the nodes and some nodes not contained in the graph. In other words, we can bound our instances from above (because the instances will always be the subgraphs of the template), but we cannot set the lower bounds. It would be logical if we could specify the required elements of the template and the optional elements.

We would also like to mention a fact that our system does not allow the user to specify the ranges of attribute values for numeric attributes and wildcards or regular expressions for string attributes. However, these two limitations can be worked around with the use of OCL constraints.

Another limitation of the system is that it outputs the instances in the DOT format. So for practical usage we have to either convert the DOT files into the format we need or implement our own Instance Interpreter which will produce the instances in the required format (which is actually rather easy).

5.1 Future work

As for the future work, we would like to say that in the current implementation of the ASMIG+ system the graph definition language we introduced has the same expressiveness as the ITG itself. So any improvements in the language would include modification of the instance generation process in ASMIG.

The ideas for the improvements of the system are mainly caused by the limitations. First of all, the full OCL support has to be added to the system. Secondly, as we mentioned before, we have to add a feature of specifying the parts of the template which are mandatory in all instances. It would also be good to add conditional expressions. For example, the node A is required in the instance if the node B is present. This would remove the problem of having an edge without any nodes.

The next logical step of developing our ITG language would be adding the ranged values and regular expressions or wildcard characters for strings. It is worth mentioning that for the numeric values this functionality is already implemented in the USE system and it does not seem a difficult thing to add.

The output format of the instances also has to be improved. One of the ideas we had in relation to this aspect is the following. If the input model corresponds to the UML Class diagram, and the output (the instance) corresponds to the Object diagram, we could instantiate the actual objects in the Java language if the user provided the corresponding classes with setters. Then these objects can be serialized and used afterwards as the test input.

Now we would like to address the wider question of applicability of model finders for test data generation. The main problem here is that we can generate the test inputs, but we cannot generate the expected test outputs. We would need a 'test oracle'. This fact makes the approach applicable only for smoke testing where we just want to check that the software under test does not crash on different inputs, and we don't care much about the outputs.

However, model finders are perfect for generating complex hierarchical data objects, and the systems that process these complex data structures have to be tested. The generated test data can help to reveal defects in the boundary cases (e.g. when some collection contains no elements or one element) which covers the most of software flaws according to Daniel Jackson. In [9] it is called the 'small scope hypothesis'.

A good example of such a complex data structure is a programming language. We think that this kind of testing with the use of instance generation would be extremely helpful in the testing of language parsers. Therefore, our work has an application and the tools performing the model finding have to be further improved to meet the needs of test data generation.

Appendix A

Test cases for the Equivalence Partitions testing of the FactoryFacade class

Equivalence Partitions for the input values of the methods of the FactoryFacade class (partitions with * denote error cases):

EP for FactoryFacade(ecoreFileName)
1*) invalid file name
2*) valid file name, invalid model
3) valid model

EP for addNode(className, nodeName)

className:
4*) not existing class
5) existing class

nodeName:
6*) invalid name
7) valid name

EP for setAttribute(nodeName, attrName, value). Testing of this and subsequent methods requires the existence of nodes in a graph. So here comes the Combinational aspect, we need to first create nodes, and then test attributes and edges.

nodeName:
8*) not existing node
9) existing node

attrName:
10*) not existing attribute
11) existing attribute

value:
12*) invalid value for Integer attribute
13*) invalid value for Boolean attribute
14*) attempt to set Enumeration value (not supported)
15) valid value for Boolean attribute
16) valid value for Integer attribute
17) valid value for String attribute

EP for addEdge(srcNodeName, destNodeName)

srcNodeName
18*) source node does not exist
19) source node exists

destNodeName
20*) destination node does not exist
21) destination node exists

22*) source and destination nodes exist, but there is no possible reference between them
23*) source and destination nodes exist, and there is more than one possible reference between them
24) source and destination nodes exist, and there is one possible reference

EP for addEdgeWithReferenceName(srcNodeName, destNodeName, refName)

srcNodeName
25*) source node does not exist
26) source node exists

destNodeName
27*) destination node does not exist
28) destination node exists

refName

29*) not existing reference
30*) reference exists, but not between these classes
31) reference exists between these two classes

We would like to make several remarks. For the partition 6 (invalid node name) we didn't previously specify the rules for valid node names. We decided to treat names without space, non-empty and starting with a letter as valid.

As for the partition 13, we decided to use method Boolean.parseBoolean() to convert an input string into a boolean. This method does not treat any input values as invalid, on all strings except 'true' (case insensitive) it returns false. So in our API the behaviour is the same - if a boolean value is set to some unrecognizable string, it is set to false.

Appendix B

Example of the DSL code for the BibTeX model

Here we would like to show how the actual generated script looks like for one of the models. We chose a model of BibTeX which is a format of bibliographic records. The diagram of the model is shown in Figure B.1. Note that many classifiers in this model are abstract (written in italic). Abstract classifiers don't get instantiated and therefore they are not present in the instance template. The following code of the template graph in our DSL was generated for this model:

```
model "BIBTEXML.ecore"

InBook inbook1
Proceedings proceedings1
Manual manual1
Author author1
Author author2
InProceedings inproceedings1
Book book1
Unpublished unpublished1
Conference conference1
MastersThesis mastersthesis1
PhdThesis phdthesis1
Misc misc1
BibtexFile bibtexfile1
Booklet booklet1
Article article1
InCollection incollection1
TechReport techreport1

bibtexfile1.entries=article1
bibtexfile1.entries=incollection1
```

```
bibtexfile1.entries=techreport1
bibtexfile1.entries=unpublished1
bibtexfile1.entries=inproceedings1
bibtexfile1.entries=misc1
bibtexfile1.entries=book1
bibtexfile1.entries=manual1
bibtexfile1.entries=conference1
bibtexfile1.entries=mastersthesis1
bibtexfile1.entries=phdthesis1
bibtexfile1.entries=inbook1
bibtexfile1.entries=proceedings1
bibtexfile1.entries=booklet1
inbook1.authors=author2
inbook1.authors=author1
phdthesis1.authors=author2
phdthesis1.authors=author1
mastersthesis1.authors=author2
mastersthesis1.authors=author1
conference1.authors=author2
conference1.authors=author1
manual1.authors=author2
manual1.authors=author1
book1.authors=author2
book1.authors=author1
unpublished1.authors=author2
unpublished1.authors=author1
article1.authors=author2
article1.authors=author1
incollection1.authors=author2
incollection1.authors=author1
techreport1.authors=author2
techreport1.authors=author1
inproceedings1.authors=author2
inproceedings1.authors=author1
booklet1.authors=author2
booklet1.authors=author1
misc1.authors=author2
misc1.authors=author1
```

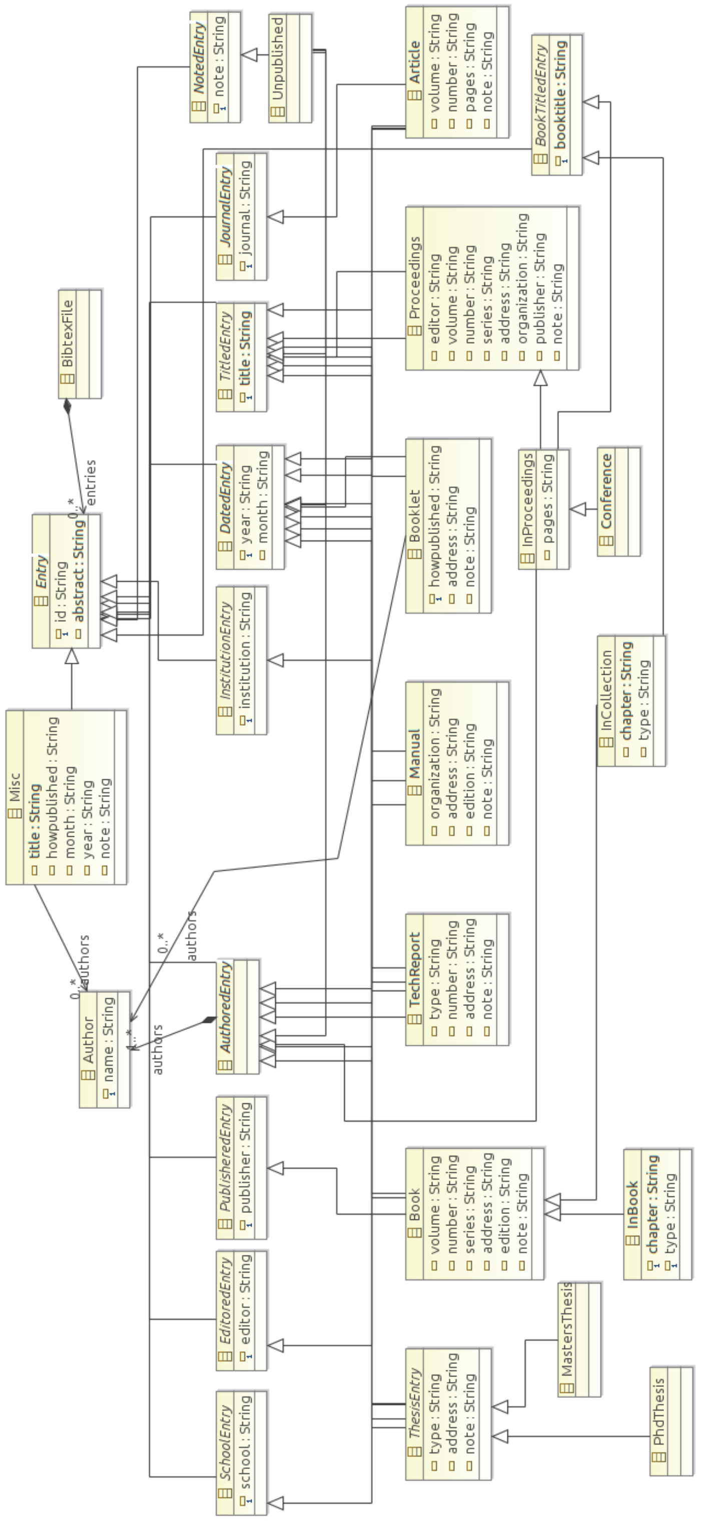


Figure B.1: Diagram for a model of BibTeX format

Appendix C

Test models set

Table C.1: Size of Ecore models used for testing.

Model	Classes	References	Attributes
A_small_example.ecore	7	9	4
BIBTEXML.ecore	28	4	55
BPMN.ecore	18	31	11
BusinessProcessModel.ecore	26	17	0
C.ecore	34	6	3
CPL.ecore	32	16	42
CPP.ecore	16	5	21
CSharp.ecore	10	10	17
Chain.ecore	1	1	0
Class.ecore	4	5	1
Company.ecore	7	9	4
CoverageTest.ecore	3	1	3
DIT.ecore	2	1	0
DOT.ecore	26	30	44
DoDAF-SV5.ecore	31	59	9
Enum_simple.ecore	4	0	5
FSM.ecore	6	4	3
FiniteStateMachine.ecore	6	13	3
GWPNV5.ecore	7	14	1
GraphColoring.ecore	1	1	1
GraphML.ecore	11	13	12
HierarchicalStateMachine.ecore	15	31	13
ICST.ecore	1	0	1
JAVA3.ecore	11	19	6

Model	Classes	References	Attributes
JavaClass.ecore	8	6	1
KM3.ecore	12	10	8
LCOM.ecore	2	1	0
MoDAF-AV.ecore	48	40	33
NOC.ecore	1	1	0
Person.ecore	1	0	1
QoS.ecore	24	30	5
RFC.ecore	2	3	0
RoyalAndLoyal.ecore	15	54	34
Sample.ecore	4	3	3
T1.ecore	3	2	1
UML2_CD.ecore	40	75	18
UML_2.0_CD.ecore	40	75	18
WebApplications_ConceptualModel.ecore	19	32	0
bi_FSM.ecore	6	8	3
bi_Some.ecore	2	2	0
bi_one2one.ecore	3	2	0
bi_one2one_name_simple.ecore	2	2	0
bi_one2one_simple.ecore	2	2	0
bi_one2some_simple.ecore	2	2	0
bi_one2zero_or_one.ecore	4	2	0
bi_one2zero_or_one_simple.ecore	2	2	0
bi_zeor_or_some.ecore	2	2	0
bi_zero_or_some.ecore	2	2	0
bi_zeroorone.ecore	2	2	0
boundtest1.ecore	6	2	0
coverage_test1.ecore	2	0	4
dsl.ecore	4	5	6
ecore.ecore	22	48	33
lightjava.ecore	32	4	2
lunar.ecore	7	6	17
mccabe.ecore	1	1	0
ocl_simple_example.ecore	4	2	2
ocl_simple_example_1.ecore	3	2	1
oclimm.ecore	2	1	4
paper2_example1.ecore	5	4	2
simplemodel.ecore	3	4	0
simplemodel1.ecore	2	2	0

Model	Classes	References	Attributes
string.ecore	1	0	3
student.ecore	8	2	1
student_nobody.ecore	7	1	1
test1.ecore	4	2	4
tree.ecore	6	0	1
umlPrimitiveTypes.ecore	0	0	0
un_Some.ecore	5	1	0
un_Some_order_simple.ecore	2	1	0
un_Some_simple.ecore	2	1	0
un_ZeroSome.ecore	4	2	0
un_one2one.ecore	5	1	0
un_one2one_simple.ecore	2	1	0
un_zero2one.ecore	4	1	0

Bibliography

[1] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1st edition, 2008.

[2] Brian Henderson-Sellers. *On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages*. Springer, Berlin, 2012.

[3] Alfons Laarman and Ivan Kurtev. Ontological Metamodeling with explicit instantiation. In *Proceedings of the Second International Conference on Software Language Engineering*, SLE'09, pages 174–183, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] Ed Seidewitz. What Models Mean. *IEEE Softw.*, 20(5):26–32, September 2003.

[5] Thomas Kühne. Matters of (Meta-) Modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.

[6] OMG. OMG Object Constraint Language (OCL), Version 2.3.1, January 2012.

[7] Sanjai Narain. Network Configuration Management via Model Finding. In David N. Blank-Edelman, editor, *LISA*, pages 155–168. USENIX, 2005.

[8] Karsten Ehrig, Jochen Malte Küster, and Gabriele Taentzer. Generating instance models from meta models. *Software & Systems Modeling*, 8(4):479–500, 2009.

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

[10] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Science*

of Computer Programming, 69(13):27 – 34, 2007. Special issue on Experimental Software and Toolkits.

[11] Martin Gogolla, Jørn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software & Systems Modeling*, 4(4):386–398, 2005.

[12] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive validation of OCL Models by Integrating SAT Solving into USE. In Judith Bishop and Antonio Vallecillo, editors, *Objects, Models, Components, Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 290–306. Springer Berlin Heidelberg, 2011.

[13] Mirco Kuhlmann and Martin Gogolla. From UML and OCL to Relational Logic and Back. In Robert B. France, Jrgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *MoDELS*, volume 7590 of *Lecture Notes in Computer Science*, pages 415–431. Springer, 2012.

[14] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer Berlin Heidelberg, 2007.

[15] A short guide to the USE Model Validator. Available at: `http://sourceforge.net/p/useocl/discussion/928881/thread/df56f303/a1d6/attachment/Usage.pdf`.

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

[17] Eclipse Modeling Framework (EMF) - Tutorial. Available at: `http://www.vogella.com/tutorials/EclipseEMF/article.html`.

[18] 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.

[19] Martin Fowler. *Domain Specific Languages.* Addison-Wesley Professional, 1st edition, 2010.

[20] Sebastian Erdweg, Tijs Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D.P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin Vlist, Guido H. Wachsmuth, and Jimi Woning. The State of the Art in Language Workbenches. In Martin Erwig, Richard F. Paige, and Eric Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer International Publishing, 2013.

[21] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* CreateSpace Independent Publishing Platform, 2013.

[22] Michael Pfeiffer and Josef Pichler. A Comparison of Tool Support for Textual Domain-Specific Languages. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, pages 1–7, 2008.

[23] Roman Stoffel. Comparing Language Workbenches. In *MSE-seminar: Program Analysis and Transformation*, University of Applied Sciences Rapperswil (HSR), Switzerland, pages 18–24, 2010.

[24] Bernhard Merkle. Textual Modeling Tools: Overview and Comparison of Language Workbenches. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '10, pages 139–148, New York, NY, USA, 2010. ACM.

[25] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 114–129, Berlin, Heidelberg, 2009. Springer-Verlag.

[26] Christopher Guntli. Create a DSL in Eclipse. Technical report, HSR-University of Applied Science in Rapperswil, 2010.

[27] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Model-Based Language Engineering with EMFText. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering IV*, volume 7680

of *Lecture Notes in Computer Science*, pages 322–345. Springer Berlin Heidelberg, 2013.

[28] Michael Himsolt. GML: A portable graph file format. Technical report, Universität Passau, 94030 Passau, Germany, 1997.

[29] Emden R Gansner and Stephen C North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11):1203–1233, 2000.

[30] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz-open source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2002.

[31] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Pearson Education, 1994.

[32] Stephen Brown, Joe Timoney, Tom Lysaght, and Deshi Ye. *Software Testing: Principles and Practice.* China Machine Press, 2012.

[33] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework.* Pearson Education, 2008.

[34] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, volume 13, page 14, 2010.