Improving the Testability of Code Generation for Microsoft DSL Tools

Testing Model-To-Code Transformations
Thesis Report

Version of January 12, 2010

Boaz Pat-El

Improving the Testability of Code Generation for Microsoft DSL Tools

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Boaz Pat-El
born in Leidschendam, the Netherlands

TUDelft

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Improving the Testability of Code Generation for Microsoft DSL Tools

Author: Boaz Pat-El
Student id: 1150464
Email: boazpat_el@hotmail.com

Abstract

There are noticeable issues related to the traditional method for software engineering, perhaps the most significant being the current cost and time to market of software systems. Microsoft's Software Factories (SF) confront these problems. A tool that supports creating SFs is the Microsoft Tools for Domain-Specific Languages (DSL Tools), which allows developers to define a modeling language, generate visual designers and transform models, that are described in custom modeling languages, to code. It does not provide any means for testing the transformation process from model to code, so that testing boils down to repeatedly executing the code generation with a variety of input. A proposed method for improving the testing process is to build a tool that provides support for testing the code generation process. In this report, we describe how we built a tool that automatically builds input testmodels for SFs, optimizes generated test sets based on predicted test quality, supplies tests to a SF and generates a log-file based on the results of the testing process. Then, several methods for qualifying the quality of generated tests are proposed, which are based on coverage criteria. After that, we illustrate the results of a case study where the effectiveness of the automated test generation approach is tested, as well as the performance of coverage criteria in predicting test set quality. Twelve errors were exposed in the SF under test, most of these being robustness errors which appear difficult to find using traditional testing approaches. In addition, testing the performance of coverage criteria indicated that there was a relation between metamodel coverage and the number of errors found and that, of all proposed coverage criteria, average metamodel coverage was the best predictor for test quality. Finally, we conclude that automated test generation is a promising approach for improving the testing process for SFs.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Drs. S.D. Vermolen, Faculty EEMCS, TU Delft
Company supervisor:	Drs. S. Bockting, Avanade

Contents

Chapter 1

Introduction

Traditionally, building software systems starts with the creation of models that describe a particular problem domain and then using these models as a blueprint to build the source code that makes up the software system. Models are primarily used as a means to communicate ideas between developers and domain specialists [22] and to structure the problem domain in such a way that it becomes comprehensible for the developers. Such (informal) models are often written in the Unified Modeling Language (UML) [52] and used as a basis for code written in a General Purpose Language (GPL) like C++ or Java.

Although this method works for designing and producing software systems, it has some noticeable disadvantages. Source code written in a GPL is generally hard for a non-programmer domain specialist to understand, implying that such a specialist will not be able to verify or understand the system or design by inspecting the code. In addition, models that are to be transformed into source code are often platform specific in terms of development and deployment [18]. This makes it hard to transport a system from one platform to another. Such models are not designed with reusability in mind, so even if the platform is the same, models are not often reusable to use in other, similar problems.

Perhaps the most significant issue with the traditional approach to designing and producing software systems concerns the cost and time to market of software systems. Software gets more and more complex, making design and realisation increasingly costly and difficult. This, in addition to the need for software developers to deliver products in increasingly shorter amounts of time, calls for new approaches for designing and producing software.

1.1 Model-Driven Development

An approach that confronts the problems of traditional software development is known as Model-Driven (Software) Development (MDD) [18]. The concept of MDD is that models drive the development process and that the level of abstraction is raised in which software systems are designed [19]. MDD's purpose is to improve developers' productivity by imposing structure and standards to the modeling process [36]. In addition, it seeks to automate the process of transforming high-level models into lower-level models or source code. This should then reduce the complexity of the software artifacts that developers use and reduce the effort to create these artifacts[36]. In particular, it should increase the reuse of design for solutions in similar domains.

1.1.1 Model and metamodel definitions

As we have introduced the concept of *Model*-Driven Development, it is also useful to define the concept of models and modeling languages. In the MDD view supported by the Object Management Group (OMG) [8], there is a four-layer architecture that forms the basis of MDD technologies [18]. Models are abstractions of a real system. They define the concepts that make up the system and describe how that system works. It is the first level of abstraction (referred to as *M1*) of a system (which is referred to as *M0*). Models can be written in an informal language, such as English, or they can be formally described using modeling languages. An example of such a modeling language is UML [13] from OMG. UML defines a model as being a definition of a system and uses constructs like classes, methods and relations for modeling real systems. As UML is a language in which models can be described, it is known as being a *metamodel* (referred to as *M2*), it is the set of constructs and rules for constructing models of a real system. UML itself, which is the second level of abstraction, is specified in the Meta-Object Facility (OMF) architecture. This architecture is an abstraction for the UML language and describes the modeling constructs of UML. Being a third level of abstraction (*M3*), a metamodel of UML, OMF is also called a *meta-metamodel* [18] although it is generally referred to as a metamodel [55]. Figure 1.1 displays the hierarchy of levels of MDD and shows what can be expressed by a level.

1.1.2 MDD promises and challenges

To illustrate what MDD should mean to the software developers, Atkinson [18] defines four fundamental changes of MDD to the traditional software development process:

1. As deadlines approach and time grows short, programmers tend to implement what comes up in their minds without writing down what they actually were considering. In a typical software organization, personnel come and go. Much information about a software system gets lost as the knowledge disappears along with the programmers. In MDD, models are considered to be important artifacts in the development process and are supposed to be accessible and useful to all interested stakeholders, including customers. This should enforce that artifacts outlive their creators, thus reducing the risks of losing information.

2. During the development of software systems, requirements are likely to change. In the traditional software process, these changes can have a large impact. Not only do the requirements and the design have to be altered, the source code is usually affected as well, meaning refactoring of the code. MDD's purpose is to reduce the effect of changing requirements by automating the process where design models are transformed into source code. When a requirement changes, this would imply that only the design would have to be changed and that the corresponding changes would be automatically placed into the source code.

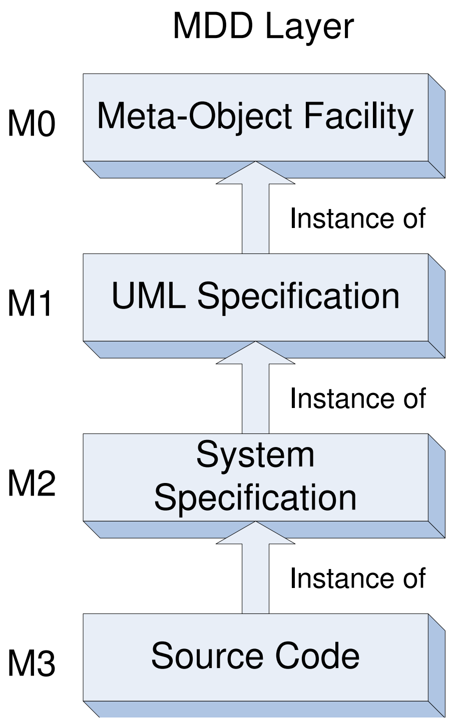


Figure 1.1: The four-layer architecture of MDD.

3. Source code or other low-level artifacts are usually tied to a tool, making the artifacts only useful during the lifetime of the tools. Such a tool could be an integrated development environment (IDE) or a compiler. One of MDD's objectives is to decouple artifacts from their development environments. For example, tools could store artifacts in such a way that they can be used by many other tools.

4. By automatising the transformation from design to primary artifact, as described in the second point, MDD should achieve that any platform-specific software artifacts should be generated from platform-independent ones.

Changes do not always come cheap, however, and the MDD approach presents some challenges for the research community. For instance, in seeking a standard modeling language, UML was proposed as a candidate [31]. UML 2.0 has properties that make it a good candidate. One of these is that UML 2.0 introduces structured classifiers and port concepts. Structured classifiers describe the internal structure of a class and port concepts specify how the internal parts of a class communicate with each other. Ports are also used to describe the interface between classes. These concepts have some properties that are related to MDD, as they allow for reusability (the design composite classes can be reused), and describe semantics that can be translated to implementations. Figure 1.2 presents an example of a UML model in which a library has been modelled using UML class constructs. Figure 1.3 shows an example of structured classifiers and ports in a UML 2.0 diagram.

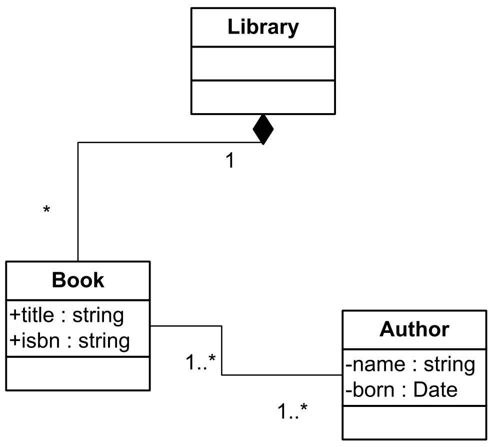


Figure 1.2: A library expressed in UML

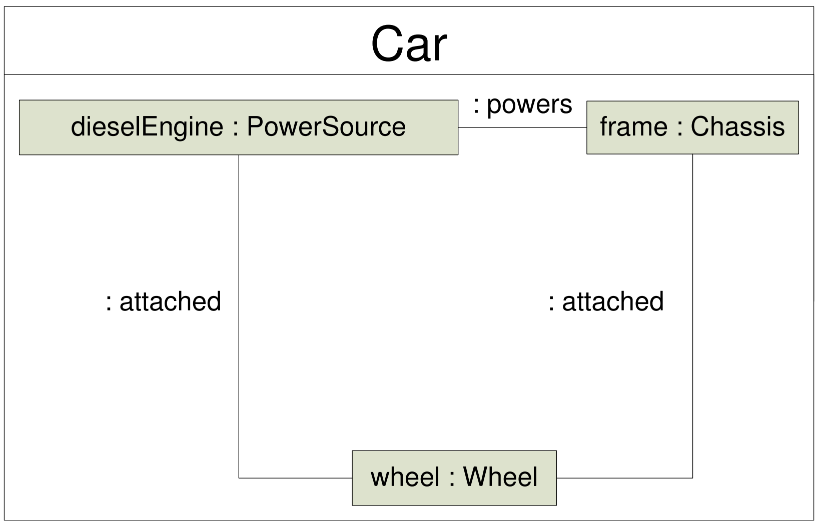


Figure 1.3: Structured classifiers express a car in UML 2.0

Regarding the semantic expressiveness of UML 2.0 diagrams, Hailpern [36] notes that:

> Unfortunately, some of the constructs in UML 2.0 are nearly semantics-free (e.g. use cases). This dearth of semantics complicates the correct usage of UML extensions, reduces their expressive power, and limits the ability of tool vendors to provide reliable, consistent model technologies.

France [31] explains that language constructs are added or extended to make up for the lack of expressive power that is needed for MDD and modeling of product families. On the other hand, he notes:

> However, the complexity of the current UML 2.0 metamodel can make understanding, using, extending, and evolving the metamodel difficult.

In addition, Berkenkotter [40] explains that, in the field of real-time software systems, high standards for software specification are required due to timing constraints and safetycritical background. He explains that UML 2.0 lacks constructs for modeling real-time behaviour due to poor time mechanisms. Also, because the UML specification is still informal, so that it in many cases is insufficient for efficient testing, which is required when dealing with safety-critical systems in a real-time domain.

1.1.3 Implementation of MDD

MDD's broad notion has been the foundation for several views on how to improve the development of software systems. One of these comes from OMG known as Model Driven Architecture (MDA) [44, 52]. Another implementation is the Software Factories from Microsoft, which will be explained in more detail in the following section. As with MDD, the main notion of MDA is to make models the primary artefacts that drive the development process [22]. MDA recognizes the similarities between many different software systems and encourages reuse and portability of the models that were used to design those systems. In particular, MDA makes a distinction between Platform-Independent Models (PIMs) and Platform-Specific Models (PSMs) [52, 35].

In contrast to the traditional approach to software engineering, in MDA software is implemented by defining models of a system at a high level of abstraction which are then converted (mapped) to more detailed models and finally into code. The more abstract models define the overall architecture of a system [18], while the less abstract ones get more specific regarding development (i.e. programming languages used) and deployment (i.e. running OS, what kind of database to be used).

The MDA approach also has some advantages with respect to the traditional approach to software engineering. The more abstract models are easier to develop, understand and verify by domain specialists who know little about software. Also, PIMs are not specific to any platform and consequently can be more easily reused in other projects where systems must be build that are in the same domain. In addition, when automated tools are used to transform the abstract models into detailed models (like source code), then changes in the architecture will have less impact on the development process.

It has been shown that, using tools that support MDA development, the MDA approach can improve the productivity of the development process compared to the traditional method of software engineering [55, 15]. Still, MDA has the disadvantage that domain specialists must have a decent understanding of the modeling languages (like UML, OCL and MOF [55, 37]), that are used to express the models of the system, to comprehend a system description. This makes it inevitable that software still must be designed and developed by software engineers.

1.2 Microsoft's Software Factories

Another popular view of MDD is a programming paradigm from Microsoft with its Software Factories (SF) [34]. SF puts the emphasis on identifying software product families and constructing frameworks to rapidly build software systems belonging to such families [35]. The idea behind product families is that many customers want the same (kind of) product, but tailored to their needs. Hence, the purpose of SF is to reduce the costs and production time of software development by means of systematic reuse to achieve mass customization [33]. As in the traditional programming methodology, UML models are primarily used to communicate ideas between stakeholders and are often informal [34]. In contrast to UML, Domain Specific Languages (DSLs) are used to model the software systems.

1.2.1 Domain Specific Languages

Van Deursen et al. [16] defined a DSL as follows:

> A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

In contrast to general purpose languages (GPLs) such as $C\#$ or Java, DSLs are restricted to their corresponding domain. Where in $C\#$ one would have to describe a system in terms of classes and programming statements, with a DSL the same system can be expressed at a higher level of abstraction.

The SF approach has some advantages over traditional software engineering. The models written in DSLs are more abstract and can be expressed in terms of the problem domain. Hence, they can be written and understood by domain specialists that have little knowledge of software development, or even be written by the customers themselves. In addition, such models can be more easily developed and verified by domain specialists. When a DSL has been developed, it can easily be reused in other projects where systems must be built that are in the same domain. Furthermore, because automated tools are used to transform the abstract models (written in DSLs) to code, changes in the architecture of a system has less impact on the development process. According to [15], SF improves the productivity of the development process compared to the traditional method of software engineering, given that the DSL for the problem domain has already been developed. However, developing a DSL is a time-consuming process that requires knowledge from both domain specialists and software engineers [45].

1.2.2 Code Generation Tools

One of the tools that has been created to support the vision of software factories is the Microsoft Tools for Domain-Specific Languages (DSL Tools) [2]. DSL Tools allows developers to define a modeling language, generate visual designers that are customized to a particular problem domain and generate code from models described in custom modeling languages. A combination of a modeling language, visual designer and code generator is called a Software Factory (SF). DSL Tools provides additional support to the developer by supplying language templates and debugging tools. Also, it provides the possibility to add certain constraints to the domain models in addition to what can be expressed in the syntax of the developed language (i.e. models created with custom visual designers can be checked for being consistent with their metamodel) [14]. Code generation is achieved by using text templates. A text template contains code that reads a model and generates a text file based on the logic that is defined by the text template. Basically, generating code happens through concatenating lines of code and placing the result in output files [14].

An example of a factory created using Microsoft DSL Tools is Avanade Connected Architectures®for .NET (ACA.NET) [5]. ACA.NET is a framework that helps developers building service-oriented .NET applications by providing services and a set of reusable architecture components that accelerate the development process. In addition, it provides a language built with DSL Tools that developers can use to built their own solutions. In chapter 2, we will go into more detail regarding the possibilities and structure of both DSL Tools and ACA.NET.

1.2.3 Model Transformations

A metamodel is a model that describes the structure of a modeling language [53]. Thus, transforming one model into another requires identifying the relationships between the models described by their respective metamodels [53]. Transforming one model into another by hand can require a lot of time and effort. Therefore, these approaches aim to automate the transformation process between models. In tools like DSL Tools and the Eclipse Modeling Framework (EMF) [27], models are transformed into source code. The purpose of source code is that a computer can execute it, while models are generally used to get an abstract view of a system. Hence, model-to-code transformations are sometimes regarded as being different to model-to-model transformations [32]. However, as source code can be regarded as a model of a system, the transformation from model to code is actually a case of model-to-model transformation [29]. Thus, generating code from a UML model or from a DSL can be regarded as a transformation of a model.

1.3 Quality of MDD Solutions

As we explained previously, one of the goals of MDD is to reduce the time and effort needed to create new software systems. One way that SF and MDA attempt to do this, is by automating the transformation process of models to source code. Still, developing automatic model transformation tools is far from trivial. One might wonder whether a software system that has been automatically generated using the MDD approach is of the same quality as software that has been produced by hand. This is a difficult question to answer, as the notion of software quality is a highly debated topic [50, 51]. Still, there are some characteristics of software that many researches agree on that contribute to its quality, one of these being the *reliability* of software systems [11, 42, 50, 51]. An important approach to software reliability engineering is the removal of faults, with software testing being one of the most important techniques for removing faults from software [42]. So in order to assert the quality of MDD solutions, we must be able to test such software systems thoroughly.

When producing a software system using MDD, models are transformed into source code. To ensure the quality of the system, a developer may choose to test the generated code using traditional software testing approaches. However, this implies that developers should, for each program that has been generated using code generating software, develop tests for code that has been generated automatically. Furthermore, one of the purposes of MDD is to decrease the implications of changing requirements. If the models that were used to generate software are changed because of changes in the requirements, tests for the generated software should be created anew. Rather, one would like to have some insurance that generated code is error-free and that additional testing is not required. Therefore, instead of testing generated code, it would seem more effective to test the generation software, so as to improve the quality of both the generation software and the solutions created using the generation software. Unfortunately, although much research has been done on using MDD for software development, many challenges remain for the process of software testing in an MDD context [29]. In the case of SF and MDA, testing strategies are needed to ensure that model transformations are correct.

1.4 Problem Statement

In the field of software engineering, much effort is put into building reliable software systems. If such systems are not able to meet up with their requirements, the system becomes unreliable, produces incorrect results or could even become unsafe. To ensure that a software system is reliable, system engineers must examine the different parts to ensure that their system works as it should. However, verifying the correctness of a software system by only inspecting the source code is generally not sufficient. To raise the reliability (and quality) of a program, software is usually tested using a wide range of testcases. Many approaches exist for testing software systems written in GPLs (for example [49, 46]). Also, methods exist for verifying models and model transformations [39, 29]. Using these conventional approaches for testing software, it is possible to test whether generated source code from a specific model is reliable and conforms to the desired specifications of the

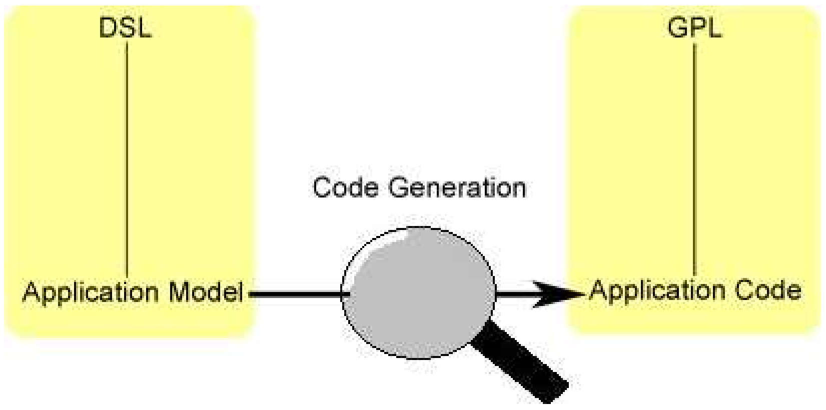


Figure 1.4: The proposed extension should test the transformation from models to code.

developer. Yet, source code generated from a slightly different model can already be erroneous because of an error in the generation process. Because tests are generally created for a specific instance of a system, new tests have to be created for testing this different model, which might take considerable time and effort.

The goal of our research is to extend DSL Tools with a tool that provides support for testing the transformation from model to code using automated model generation. The tool should improve the quality of the code generation and decrease the time of the test process, and hence reduce the time needed for developing DSLs. Figure 1.4 illustrates where in the development process of SF solutions, the proposed extension should be added.

The research questions are:
Can the testability of code generation using Microsoft DSL Tools be improved by automated test generation?
If so, what kind of errors can we find in generation code?

To be able to answer these questions, we build a tool that can generate models for DSLs created using DSL Tools. To evaluate the added value of using a automated test generator for testing code generators, we use our tool on ACA.NET, a DSL Tools solution created by Avanade.

The evaluate whether the tool actually does improve the testability of code generation, we attempt to determine the following:

- Do we find errors in ACA.NET that were not exposed earlier? Are more errors exposed when using the tool? Errors can be found in text templates that have already been successfully used in other projects. Any new errors that show up indicate that the testing tool for model transformations actually is an improvement over current manual testing.

- Do we find errors in ACA.NET that are still present in a later version of ACA.NET? If the tool is able to find errors in ACA.NET that are still present in a later version, then this suggests that the exposed errors are difficult to find using traditional testing approaches. To answer this question, the tool for testing DSL Tools solutions is used on both an older version of ACA.NET and a newer one. If errors are found that continue to exist through different versions of ACA.NET, then we can assume that these errors were not found by traditional testing techniques. This would be an indication that the tool for testing DSL Tools solutions adds value to the testing process.

To evaluate what kind of errors we find in the code generator, we make the following distinction in errors:

- Errors that occur during the generation of code. These are errors that occur during the transformation from model to code, such as exceptions being thrown as certain models are transformed. These errors indicate that not all valid models can be generated into source code.
- Errors that occur during the compilation of code. These are errors that occur after the model has been transformed to code and is being compiled as source code. Any compilation errors that occur may indicate that there are some valid models that do not transform correctly into code.

In addition to the moment that an error occurs, we are also interested in looking at the step from which the error originates. For example, if there is an error during the compilation process, then this could well be the result of an error in the code generation step which produced code that contained syntactical errors. Also, an exception might be thrown during the code generation step because of an error in the validation step, which results in an invalid model being transformed as if it was a valid model.

Therefore, in addition to the above two distinctions, we categorize the errors in the following way:

- Errors that originate from the validation step. Models that were supposed to be rejected before being attempted to be generated into code, but aren't, indicate that there are errors in the validation code.
- Errors that originate from the generation step. If during compilation an error occurs, then this can likely be attributed to an error in the code generation process. However, it could be an error in the validation step as well.

In chapter 5, we will go into more detail regarding the experimental setup of the research.

Furthermore, we require that it should be feasible to use the proposed tool within a typical DSL project. That implies that the tool should be easy to use in conjunction with a DSL Tools projects and that it should take (relatively) little time to use the proposed tool on real-world SF projects.

1.5 Avanade

The project of developing a tool that improves the testability of code generation will be done at Avanade. Founded in 2000, Avanade is a joint venture between Accenture and

Microsoft and has over 9000 employees from all over the world. Avanade is a global IT consultancy, serving customers in more than 20 countries worldwide, dedicated to using the Microsoft platform to help enterprises achieve profitable growth through solutions that extend Microsoft products. Avanade has a comprehensive portfolio of IT and business solutions and has deep industry experience, technology, assets and proven implementation approaches. Avanade provides solutions in the following fields:

- Application Development: Gain advantage over the competition by using custom application solutions to drive business processing and data. Avanade delivers custom enterprise applications based on the Microsoft .NET Framework. With ACA.NET, Avanade has extended the .NET platform tools, enabling teams of Microsoft Certified Solution Developers to accelerate time-to-market, dramatically reducing enterprise application development costs.
- Business Intelligence: Craft a BI strategy that fits your unique business requirements, integrates with your existing infrastructure, and aligns with your companys goals. Uncover fundamental business value from your corporate data and transform it into a strategic business asset with Avanade® Business Intelligence (BI) solutions.
- Enterprise Collaboration: Evolve your business workplace to the next generation, using Microsoft®-based technologies, where people, processes and technology are connected using collaboration tools that fuel information-sharing and productivity, driving value, profitability, and innovation.
- Infrastructure: Avanade's business expertise, technology and assets will set up Microsoft®-based infrastructure solutions to maximize existing investments, including line-of-business applications, with Avanade's business expertise, technology, and assets.
- Managed Services: Keeping your business-critical software uprunning and up-to-date is of major importance for any business. Avanade Managed Service support, maintain and evolve your enterprise business software.
- Microsoft Dynamics: It is vital For all businesses to be able to effectively maintain customer relations. Avanade has expertise in designing, implementing, and deploying Microsoft Dynamics solutions to optimize any company's return of investements.

With the Avanade Connected Architecture® (ACA®.NET) framework, Avanade has extendend the Microsoft .NET Tools framework. Since version 5.1, ACA.NET includes a DSL for designing and implementing applications. This DSL has been developed using Microsoft DSL Tools and has been used in multiple projects all over the world. The scope of ACA.NET is beyond what is now available in the university's database of software, and will provide us with an interesting and sophisticated tool for us to analyse. Therefore, it can be used as a case for determining the effectiveness of a tool that should help test the code generation process of a DSL Tools project. In addition, Avanade has seasoned developers of the ACA.NET framework readily available to provide us with insight regarding our research and our findings. For these reasons, I have chosen to perform the research at Avanade Netherlands.

1.6 Outline Thesis

In this thesis, we describe the process and results in realizing a method for improving the testability of code generators by creating an automatic model generator for DSL Tools solution. In chapter 2 we will describe DSL Tools, the tool which allows a developer to create software factories. In addition we will describe ACA.NET, a solution built using DSL Tools, as it will be used as the experimental subject for the model generator. Chapter 3 explains how the model generator is built and how it works. In chapter 4 we will illustrate what coverage criteria can be measured using the model generator and what criteria we used for our experiments. Chapter 5 demonstrates the experimental setup and subsequently shows the results of using the model generator on ACA.NET. Then, in chapter 6, the results from the experiments are discussed. In chapter 7 we explain what we concluded from the research and give some directions into possible future research regarding our findings.

Chapter 2

Background

Microsoft DSL Tools is a SF that consists of a visual designer and a DSL that allows developers to create SFs. DSL Tools has been used to create the ACA.NET SF, which is the SF that has been tested in this thesis. In the first part of this chapter, the DSL that is used by Microsoft DSL Tools will be described. Then, some research is presented regarding the testing of model transformators (e.g. code generators). The results from this research has been used to come up with a way to improve the testability of DSL Tools solutions. Appendix **??** describes ACA.NET 5.1, a SF developed by Avanade using Microsoft DSL Tools.

2.1 Microsoft DSL Tools

In this section, the language for creating DSLs will be covered first. Then, the transformation from a DSL description into XML will be described. Finally, we describe how artifacts can be generated from models written in a custom DSL

2.1.1 DSL for creating DSLs

.

The purpose of DSL Tools is that it provides a DSL and corresponding tools for creating visual DSLs [25], associated with a *DSL Application*. The DSL Application allows developers to create models using the DSL language and to generate code from those models. The DSL to create DSLs itself is a graphical language, where each concept is represented by a figure, text or line. In this section, the DSL that is used for creating visual languages will be explained, as it is described in [3] and [25]. The DSL is a object-oriented meatmodeling language and consists of two parts: a metamodel (or domain model) and a graphical notation. First, the structure of the domain model is explained. After that, the graphical notation will be covered.

Metamodel

With DSL Tools, a developer can create a DSL for modelling systems and behaviour of a certain problem domain. The DSL is a metamodel (or domain model) which represents concepts that belong to the problem domain. Each concept in the problem domain will have a corresponding concept in the domain model of the language. These concepts are depicted by domain classes (or elements within the domain model).

Also, a domain class can have multiple relationships with other domain classes, called domain relationships. In the DSL Tools language, there are three different types of domain relationships:

1. The embedding relationship; An aggregration that specifies that a concept has a life-span included in that of another.

2. A reference relationship that is the association between two concepts.

3. The inheritance relationship that specifies that one concept is a generalization of another concept. In DSL Tools, only single inheritance is allowed[1].

   To indicate that a certain domain class should not be instanced in a model, it can be declared to be *abstract* and is assumed to have one or more inheritance relationships with other, non-abstract classes. Or, to indicate that it should not be possible for a class to have subclasses, a domain class can be declared to be *sealed*, to denote that a concept cannot be a generalization.

Each domain relationship has two domain classes that are bound to it: a *source* domain class and a *target* domain class. In an embedding relationship, the target domain class has a life-span included in that of the source domain class. In an inheritance relationship, the concepts of source and target do not have any meaning, as such relationships are implicit and cannot be instanced in a model. In addition, a domain relationship defines *multiplicities* for its source and target. Multiplicities specify the ranges for the number of relations that can be bound to a source object (instance of the source class) or target object in a model. So, a domain relationship can enforce that a model is only valid if a source object is bound to at least one instance of the relationship. In DSL Tools, there are four multiplicity types: zero-to-one ($0..1$), one (1), zero-to-many ($0..*$) and one-to-many ($1..*$).

All domain classes must have at least one embedding domain relationship with another class, representing that all objects must be embedded inside another object. There is only one exception to this rule: the *RootClass* of the model cannot be embedded in another object. Instead, all classes, except for the RootClass, are directly or indirectly embedded in the RootClass. This RootClass represents the *diagram*, or *design surface*, of the editor of the DSL Application. In a DSL Application, this corresponds to all model-objects being embedded in the diagram.

Furthermore, domain classes, as well as domain relationships, have domain properties. Domain properties are similar to class attributes in a UML class diagram. A domain property can be of any type, including primitive types (e.g. integer, string) or complex types

[1]In single inheritance, a class can only have one superclass.

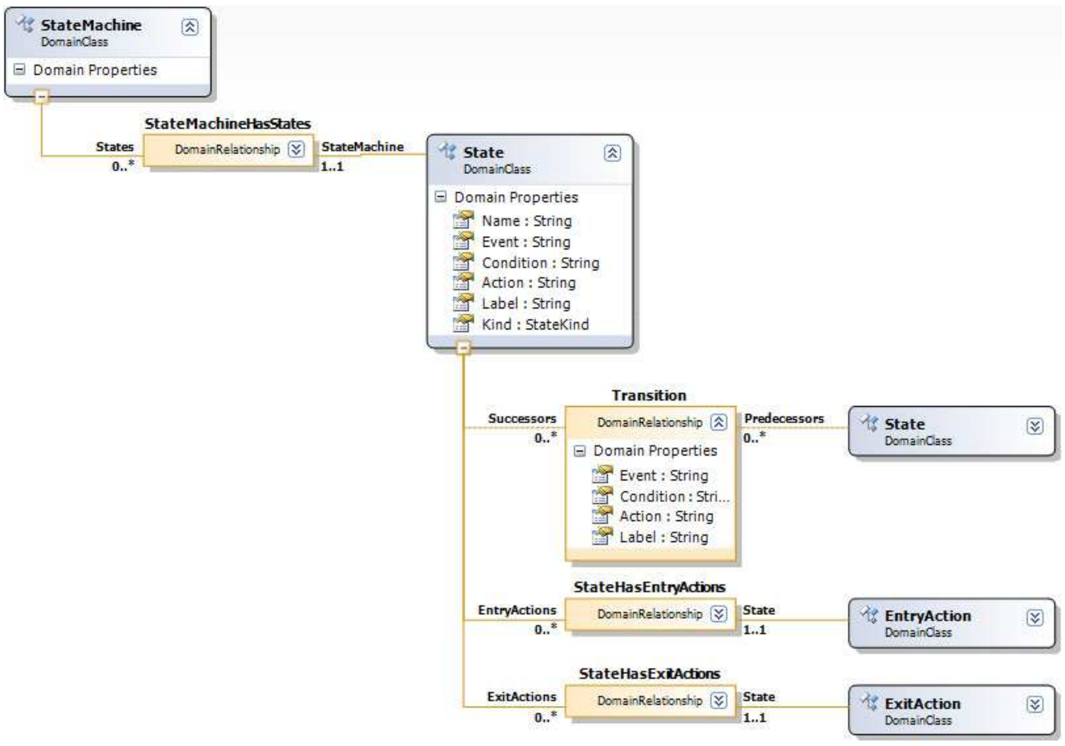


Figure 2.1: Domain classes of a state machine

(e.g. list, dictionary). A domain class inherits all domain properties of its superclass. Figure 2.1 presents an example of a state machine that has been modelled using domain class constructs.

Graphical Notation

The graphical notation describes how concepts and relationships in the domain model are represented graphically by shapes. A domain class can be represented in the graphical notation by one of three different concepts:

1. The Geometry Shape, that implies that a class is represented as a geometrical shape.
2. The Image Shape, which is used to represent a domain class as an image.
3. The Compartment Shape, that represents a class as a geometrical shape that has compartments.

Figure 2.2 shows an example of how domain classes are connected to graphical concepts in DSL Tools.

Shapes are placed in a different area of the designer than domain classes. The areas are divided by a *swimlane* that divides the designer screen into the domain model part and the graphical notation part. Swimlanes are horizontal or vertical lines on the diagram

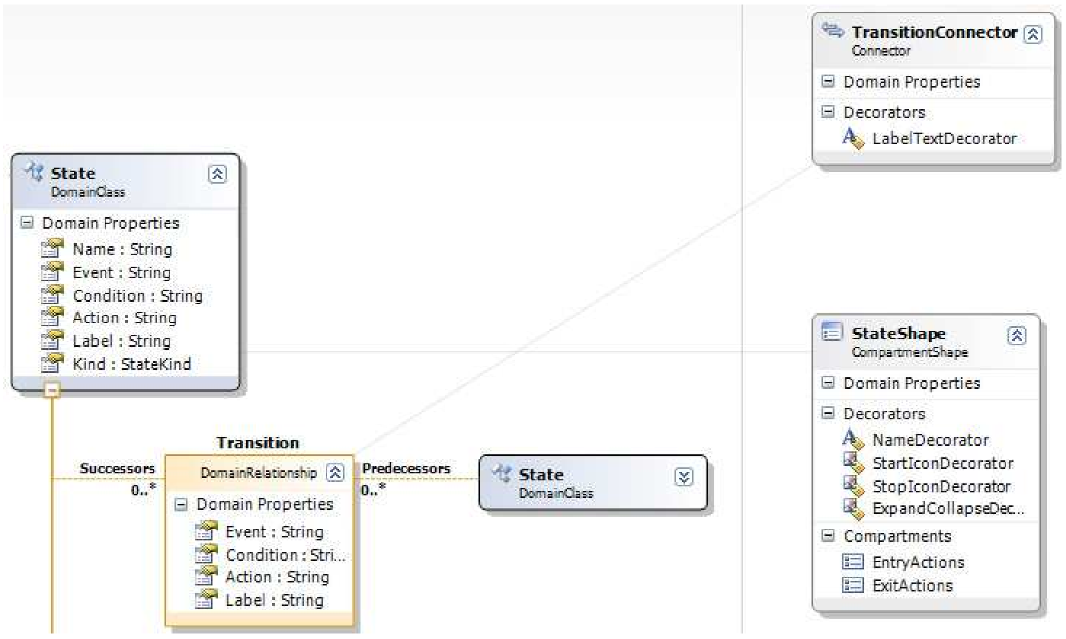


Figure 2.2: Domain classes connected to their graphical representation

that represent different areas in a diagram. The connection between the metamodel and the graphical notation is called the Diagram Element Map. The diagram element map represents the relationships between the metamodel concepts and the graphical concepts. Thus, a connection between a domain class and an image shape implies that, in the language, that particular domain class will be represented as an image.

In a DSL Application, a developer can create models consisting of one or more domain objects, which are represented by shapes. To enable a developer to select a domain object and add it to the model, the DSL Application provides a *toolbox*. The toolbox contains a number of domain classes and relationships of the metamodel. These domain classes and relationships can be selected by a developer and moved onto the design surface so as to add a selected domain object or relation to the model. Just like the design surface, the toolbox is a standard part of any DSL Application. In DSL Tools, the designer of a DSL Application can choose to add certain domain classes and domain relationships to the toolbox.

2.1.2 Customizing Model Validation

In DSL Tools, the author of a DSL Application can add constraints so as to ensure that a model built in a DSL is valid. For example, an author could enforce that a model does not contain multiple classes with the same name, or that a class only has one relationship with any other class. To enforce these restrictions, DSL Tools provides an API that enables the author to test the model for any invariants to ensure that design-time constraints are met. Constraints can be applied to a DSL by adding *rules* to the metamodel. Rules are associated with domain classes and execute when an instance of a class changes. We call these rules

Custom Validation Code (CVC), which are written in either C# or Visual Basic.

Design-time constraints come in two flavors: hard constraints and soft constraints. Hard constraints are those that the tool prevents the user from ever violating. On the other hand, soft constraints, are allowed at some moments, but not at others. For example, a constraint that enforces that no two classes in a model have the same name could be either a hard constraint or a soft constraint. If it is a hard constraint, then a user will not be allowed, at any point in time, to create a class that has the same name as another class. If it is a soft constraint, then a user can be allowed to create a class with the same name as another class, but will not allowed to safe the model to disk as long as two classes in the model have the same name. The motivation behind the distinction between hard- and soft constraints is that of usability. Enforcing hard constraints generally requires a lot of calculation and also enforces users to implement their model in a specific way. On the other hand, having too many soft constraints could result in a lot of errors coming up at the moment a developer wants to save a model to disk.

DSL Tools does not provide support for checking runtime constraints.

2.1.3 Generating Code from Models

The purpose for designing a (visual) language is to generate artifacts from models written in that language. These artifacts are text files, which can be of any type, including configuration files, content of databases and source code [25]. There are multiple approaches with DSL Tools to transform models into text files, one of them being text-templating [25]. Because, in many projects, models are transformed into source code, we'll use the term code generation, instead of artifact generation. In the following paragraphs, we explain the basics of text-templating and illustrate how code can be generated from a model using this technique.

As explained in the previous section, models written in DSLs are serialized as XML documents. As such, they can be tranformed into code by using the XSLT (Extensible Stylesheet Language Transformation) [10] language. XSLT makes use of Xpath, a syntax for defining the parts that make up an XML document, and path expressions that make it possible to navigate through an XML document [9]. In addition, XSLT provides templates that define how certain XML-parts should be translated into other languages such as HTML. However, models that are designed using a visual editor are already loaded into memory. This makes the parsing of the serialized model unnecessary as DSL Tools provides a domain-specific API to access models in memory. Making use of this API, a developer can choose to generate code via text-templating instead of via XSLT.

Text-Templating

The Text Templating Transformation Toolkit (T4) provides an alternative method for generating code from models. It is a text transformation technology developed by Microsoft and used mostly for code generation by several Microsoft products [1]. In some aspects, text-templating resembles transformations via XSLT. Text-templating makes use of text files, called text templates. Like with XSLT, a text template contains code that navigates through a model and has code that generates a text file based on the logic that is defined by the template. The text template contains two parts, one part that defines the structure and dynamic behaviour of the text template, while the other part contains the text that will be written directly to an output file. Two important differences between text-templating and XSLT is that text-templating does not navigate through an XML document, but rather through a model in memory. As such, the logic needed to navigate through a model can be defined in either C# or Visual Basic. Text-templating is Microsoft's method of choice for generating source code from models [25].

The logic required for navigating through a model is defined in a text template inside control-markers `'<#...#>'`. Within these control markers, different types of directives and control blocks can be stated. One of these control blocks is the Standard Control Block, which introduces control statements into a template. Control statements are code, usually written in $C\#$ or Visual Basic that control the flow of processing in the text template. It enables the author of the DSL to traverse the model and specify how certain patterns of a model should be transformed into code. Below is an example of control statements inside a standard control block.

```
<#
   for(int i = 0; i < 10; i++) {
      WriteLine("" + i);
   }
#>
```

Another type of control block is the Class Feature Control Block, which adds methods to the template. This type of control block is used to add reusable pieces of template. These control blocks must be placed at the end of a template file.

```
<#+
   private string addNumber(string s, int n) {
      return s + ", " + n;
   }
#>
```

In addition to the standard control block and the class feature control block, a text template can contain Expression Control Blocks. An expression control block evaluates the expression it contains and calls the .NET method ToString() on the result.

```
<#
   foreach(MyObject o in this.MyList.MyObjects) {
#>
         <#= o #>
<#
   }
#>
```

Another type of control block is the Directive Control Block that provide instructions to the templating engine. For example, directives could allow an author to split up a text template into multiple files or use classes from another assembly.

```
<#@ Assembly Name="System.Core.dll" #>
<#@ Import Namespace="System.Collections.Generic" #>
<#@ Import Namespace="System.Linq" #>
```

Additional information regarding text-templating can be found in [6].

Text-templating actually boils down to identifying patterns of a DSL and transforming these patterns into code. An advantage of using Text templates instead of XSLT is that it provides functionality that XSLT lacks, like allowing methods to be used. A weakness of using XSLT or text templates is that they lack any support for validation of verification of the transformation process. That is, there is are no constructs or tools that help determine whether there are errors in the code that transforms models into code or other artifacts. Testing boils down to repeatedly executing the code generation with a variety of input.

In the next section, we describe some research that has been done on testing software that transforms models into code.

2.2 Related Work

2.2.1 Introduction

Testing strategies for new software engineering approaches require new testing strategies. Manually creating a test set that covers all possible language and behavioural constructs can be quite challenging. Manual test cases can contain errors, therefore manually created tests must be validated as well. However, creating test cases for model transformators can be particularly difficult, because the parameters for procedures being tested (e.g. sentences of a language) are very complex [58]. Therefore, instead of only manually creating test cases, automatically generating test cases could aid the testing process greatly. Much research has been done on automatically generating test sets for software testing.

However, testing model transformations is not a new problem. Compilers are software systems that translate source code into executable code. A compiler takes a source program as input and generates an output program if the source program was valid, or error messages if the source program was invalid. Compilers have a lot in common with tools where high-level models are transformed into source code or similar artifacts. Thus, we can regard input source programs as models. Compilers are complex programs and checking the conformity of a compiler to its specifications is a complex task [23]. In addition, models usually are complex data types and therefore, creating test cases for compilers can be challenging. Instead of only manually creating test cases, automatically generating test cases could aid the testing process greatly. Predictably, automatically generating input data for testing compilers has been the subject of researches for many years [21]. During our research, we observed that some approaches that are related to testing model transformations are also used for testing compilers.

2.2.2 Automated Model Generation

Much research has been done on automatically generating test sets for software testing and, from our research, we identified multiple activities concerning the process of the automatic generation of test models, in addition to those discussed in [24]. In Figure 2.3 these activities are shown. Many approaches to model generation choose to only incorporate a subset of these activities into an approach.

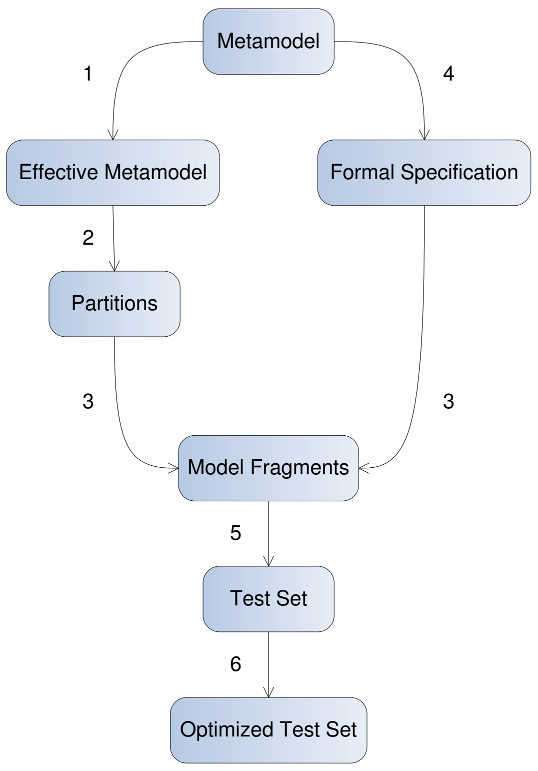


Figure 2.3: Parts of the process for automatic test set generation

1. The first step of automatic model generation often starts with identifying what parts of the model are important for the transformation that is done by the compiler under test. This important part of the metamodel is called the *effective metamodel* [24]. An effective metamodel is the part of the metamodel that actually is transformed by the compiler; Metamodel elements that are not used in the transformation process are ignored and will not become part of the effective metamodel.

2. From the metamodel, *partitions* [24] are defined. Partitions are primitive parts of the metamodel that can be used as testing data for creating oracles. For example, regarding the UML metamodel, class attributes and relations can be regarded as partitions.

3. From the (effective) metamodel, *model fragments* [24] are identified that form the smallest parts of the generated models. The model fragments are the significant parts of the metamodel. The reason for partitioning the metamodel into model fragments is to minimize the amount of non-relevant test cases in the generated test set and to optimize the test data generation process.

4. Not always are metamodels divided into partitions or fragments. In some cases, the metamodels are rewritten into other formal declaration languages. Models are then automatically generated by using some properties of the language. For example, it is possible to declare a metamodel in a *Abstract Statemachine* (ASM). An ASM is a type of state machine in which it is possible to describe algorithms on a higher level of abstraction than in an non-abstract statemachine, which makes an ASM more readable and scalable [17]. ASMs have a property that all possible states and transitions can be identified and traversed using an algorithm. This propery can be used to generate a set of test models.

5. This is the step where test models are actually generated. This step can incorporate the generation of, according to the metamodel, valid models, invalid models or both.

6. Test sets can contain many models that are similar to each other and cover the same contructs and constraints of the metamodel. As these models do not provide any additional coverage of the metamodel, they are useless and only increase the size of the test set and therefore increase the amount of time needed to perform all tests from the test set. Therefore, some approaches perform an optimization step that identifies what test cases of the test set can be left out of the test set so that no coverage is lost.

From the steps illustrated in figure 2.3, a method was chosen for automatically generating test cases for DSL Tools solutions. In this thesis, an automated test generation approach has been tested that is based on the generation of models from a metamodel definition. The following chapter describes our approach for automatically generating models from DSL definitions.

Chapter 3

Model Generator for DSL Tools

3.1 Introduction

The purpose of this thesis is to determine whether the approach of automated test generation improves the testability of code generation tools. For this reason, an automated test generation program has been created. In this chapter, our approach to generating test cases for DSL Tools solutions will be described. The software that automatically generates test cases is called the *model generator*. To be able to generate model testsets from a DSL solution, the model generator performs the following steps, which are illustrated in figure 3.1:

1. The model generator reads a DSL Tools metamodel definition to load the DSL into memory.

2. It generates random models from the DSL description.

3. Depending on metamodel coverage criteria, a subset from the generated models is selected. A justification and the details of this step are explored in more detail in chapter 4.

4. The generated test models are written to file, as is the data regarding the coverage of the test set.

Finally, the models are supplied to a DSL Application (in this case, ACA.NET), which loads the models, validates the models, generates code from them and finally compiles the generated code. If any errors occur during one of these steps, information regarding the error, and the model involved in the error, are written to file. This last step is not part of the model generator itself. Rather, it is part of a seperate program, called the *code generator tool-chain*.

The rest of this chapter gives a more detailed view of the different steps involved in automated test generation. First we describe how the DSL definitions are parsed that are generated by DSL Tools. Then the strategy for generating models from the DSL definitions is explained. After that, we explain how the behaviour of the generator can be customized

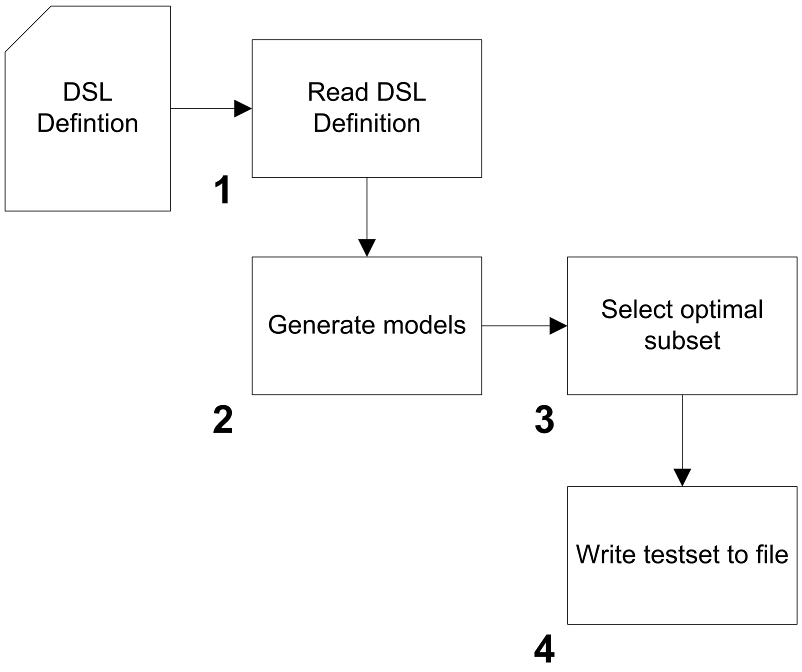


Figure 3.1: A four-step process for automatic model testset generation.

by using *generator hooks*, pieces of code that can be referenced from the model generator in order to customize its behaviour. Subsequently, an explanation is given of how the model generator chooses a subset from all generated models so as to create an optimized model testset. Finally, we describe how we created the chain where models are generated, subsequently supplied to ACA.NET, transformed into code and compiled.

3.2 Generating models using a model generator

The aim of the project is to create a model generator that is able to generate models from any DSL Tools project. Therefore, the generator must be able to interpret the definitions of the DSLs that are created using DSL Tools. In this section, we describe what data from the DSL definition we use to be able to generate models. Then we describe the strategies used to automatically generate models from a DSL definition.

3.2.1 Preparing for Model Generation

To generate models from a DSL definition, we need to perform preliminary actions. First we have to read a serialized DSL from file. Then we should interpret the definition so that we can use it to generate models for it.

Reading the DSL Definition

The first task of the model generator is to load the definition of a DSL into memory. With each DSL created using DSL Tools, this tool also generates code for parsing model-files for that particular DSL. Still, we have chosen to implement a custom DSL definition reader for our model generator. That is because the parsers that DSL Tools provide are DSL specific and the metamodeling DSL is proprietary. Our tool has to be able to parse any language definition, which makes DSL Tools' generated code not sufficient for our purposes.

The DSL definition reader works as follows. The reader starts by reading the metadata from the DSL. This supplies the generator with information regarding the name of the DSL, the version of the DSL and extension of the files in which the models are stored (in the case of ACA.NET, the extension is '.acanet'). The core of the DSL comes next, which is the collection of domain classes, domain relations and domain properties. After that, the reader reads the custom types that are used in the DSL. Now we have enough data to generate models from the DSL definition. However, in addition to generating models, the model generator must also be able to serialize these models to files in such a way that a DSL Application is able to interpret the files. The XMLClassData in the DSL definition, describes how models are supposed to be serialized to files. The final step of reading the DSL Definition file is to parse this data. After all the data has been parsed, we need to interpret the data so that we can use it to analyse the DSL and generate models for it.

Interpreting the DSL definition

After reading the DSL definition, the parsed data is made ready for use by the modelgenerator. For efficiency of the model generating algorithm, we perform the following two steps:

1. For each domain class in the DSL, make a list of the domain class' subclasses.
2. For each domain class in the DSL, determine which domain relationships it derives from a superclass (if any).

To illustrate the second step, we provide a figure of a small DSL 3.2, which has three classes: A, B and A_child. Class A is the superclass of class A_child, so that A_child inherits all properties and relationships of its superclass. Class A has a reference relationship with class B, which implies that there are two possible relationships.

With this, all preliminary actions have been done. The model generator can start generating models.

3.3 Strategy for Generating Models

Based on the properties of a DSL created in DSL Tools, it is possible to define an approach for creating a set of models for testing a code generator. The DSL consists of two primary structures: domain classes and domain relationships. Domain types are always a property of either a domain class or a domain relationship. So the key to generating models is to

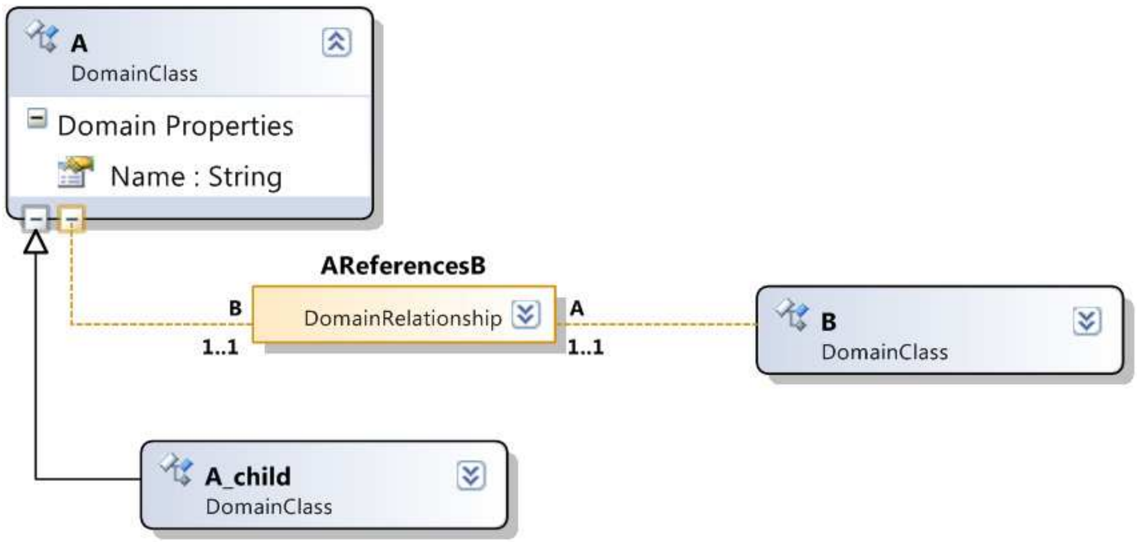


Figure 3.2: A DSL with an inherited relationship.

generate a graph of objects and relations, in such a way that a model covers as many parts of the DSL as possible.

If we regard all domain classes in a DSL definition as nodes, and the domain relationships between the classes as edges that connect the nodes, then the domain classes and the embedding relationships form a connected rooted graph[1]. Thus, there exists a path from the root class to all other domain classes, with that path made up of embedding relations. Figure 3.3 shows this idea. The algorithm therefore starts by creating a root-object and building embedding relations from the root. Each new embedding relation requires a new object to be instanced, as each embedding relation requires a unique target. From the new objects, new embedding relations are formed. Following this approach, all possible domain classes can be instanced, as all domain classes can be reached from the root. Figure 3.4 illustrates the approach, which we will describe here step-by-step:

1. Each model definition starts with a root-object, which is an instance of the domain class which has been declared as the root in the DSL. Creating an instance of the root class is the first step. Each time a new object is created by the generator, all properties of the object are instanced as well.

2. The second step is to generate any required relationships as imposed by the DSL definition. For instance, an object might have a one-to-many reference relationship with the root. This implies that the model generator should create a new reference relationship between the object and the root-object, otherwise the model will not be valid and hence cannot be used as a test case. Each time a new relation is created by the generator, all properties of the relation are instanced as well, similarly to creating a new object.

[1]A rooted graph has one node which is labelled so as to distinguish it from the graph's other nodes.

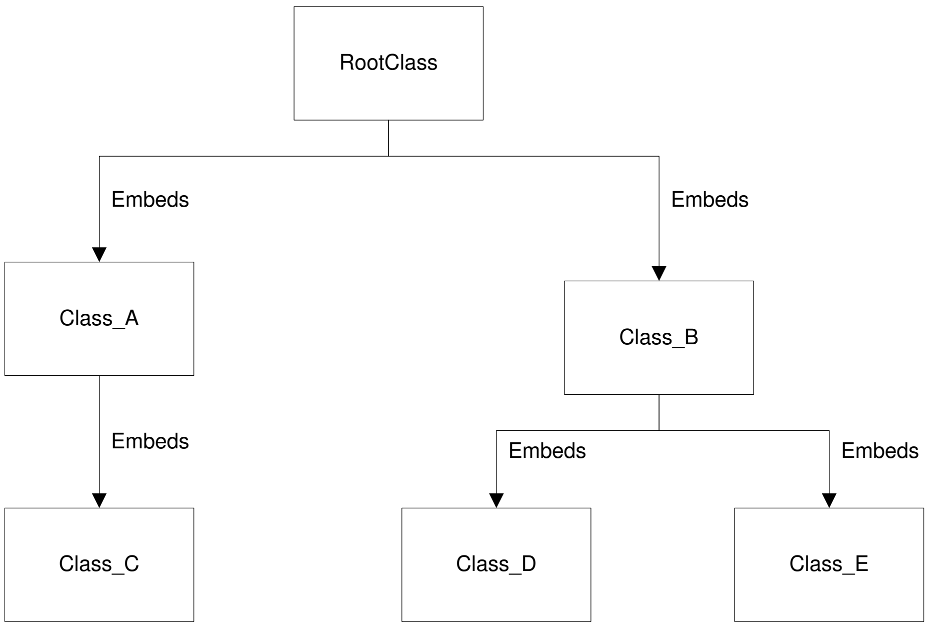


Figure 3.3: From the root, all domain classes are reachable via their embedding relations.

3. Check if a new object must be created for the generated relationships. Each embedding relation requires a new object to be created, as an object must be the target of exactly one embedding relation. In addition, it could be that, in the previous step, no suitable object was found to be the target of the new relation. Therefore, if in the previous step, an embedding relation was instanced, or no suitable target was found for a reference relation, then the next step is create a new object and go to step 2. Otherwise, continue to step 4. At the moment that the generator chooses to create an object, it already knows what type of object should be instantiated. It knows this by looking at the source or target of the relationship that was created in the previous step, and identify what domain classes are connected by the relationship. However, instead of creating a new instance of the domain class, the generator checks if the domain class has any subclasses. If so, the generator randomly chooses to instantiate the domain class or one of its subclasses. Abstract classes are never instantiated.

4. The generator counts the number of relations and the number of objects in the model. If the number of relations or objects is equal to, or exceeds the required number of relations or objects, then the algorithm stops, otherwise continue go to step 5.

5. The fifth step is to randomly pick an object in the model, and randomly create a new embedding or reference relation, where the source of the relation is the selected object. If no embedding or reference relationships are possible from this object, we choose another object to create a random relationship with. The chance of creating either a reference or an embedding relation is 50% for both. If a new reference relation is generated, the generator searches the model for all objects that can be a valid

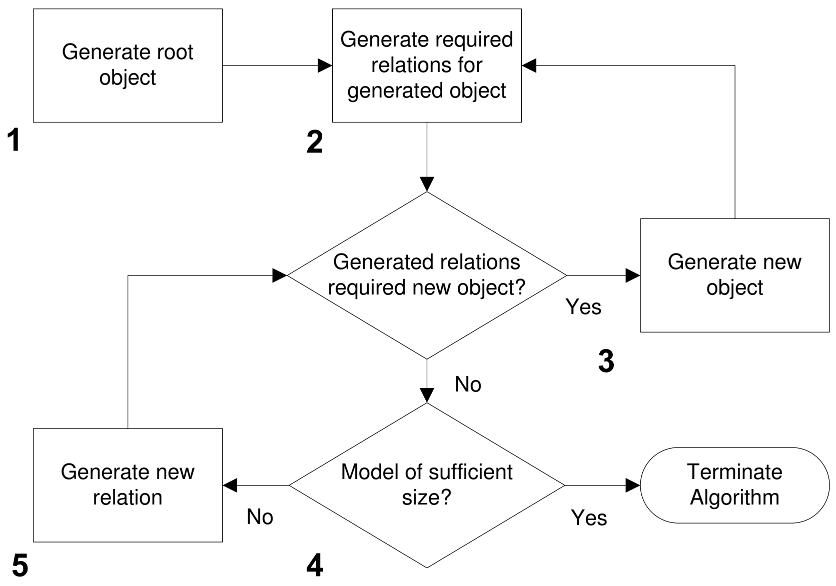

Generate root object
1
Generate required relations for generated object
2
Generated relations required new object?
Yes
Generate new object
3
No
Generate new relation
5
No
Model of sufficient size?
4
Yes
Terminate Algorithm

Figure 3.4: Algorithm for generating random models.

target of the relation and picks one randomly. If no valid targets are available, we go to step 3.

The algorithm revolves around creating relations. Objects are only instantiated at the moment that it is necessary to create a new object for a relation. An advantage of this approach is that, each time the algoritm reaches step 4, we are sure to have a valid model and are free to stop the algorithm. An alternative to this approach is to focus the algorithm on the domain objects instead. In such an algorithm, the generator starts by generating several objects, and subsequently connect these objects together by their the embedding relations. When the objects are connected, the generator starts to randomly create reference relations between the objects. The advantage of this approach would be that the tester would have more control over the amount of objects that are generated by the model generator. However, in practice this proves not to be the case.

3.4 Custom Hooks

In the previous section we presented an algorithm that is able to automatically generate models from a DSL definition. According to the DSL definition, all these models are valid and should be accepted by the corresponding DSL Application. Unfortunately, for an application created using DSL Tools, that is not the case as DSL Tools allows a DSL Application to have *custom validation code* (CVC). CVC analyses a model in memory and has the ability to reject a model as being invalid. Unlike the DSL definition and the constraints derived from the definition, CVC is imperative instead of declarative as it is written in C# code. Hence, it is extremely difficult to automatically derive what constraints are imposed by

CVC. When generating algorithms, it is important to take into account any constraints that are imposed by the CVC, in addition to those imposed by the DSL definition. Let us present an example that illustrates why this is necessary. Given the following DSL:

- The DSL has four classes: Class_Root, Class_A, Class_B and Class_C.
- Five relationships: an embedding relationship between Class_Root and classes Class_A, Class_B and Class_C. In addition, a reference relationship between Class_A and Class_B and a reference relationship between Class_A and Class_C.
- Class_B and Class_C have a domain property *Name* of the string type.

Let A denote an instance of Class_A, B denote an instance of Class_B and C denote an instance of Class_C. CVC of the corresponding DSL Application could enforce that there may only be a reference relationship between A and B if the *Name* property of B starts with *"start_with_this_string"*. Also, a relationship between A and C can only exist if the *Name* property of C starts with *"start_with_another_string"*. Then CVC could enforce that, if there are two relations between A and B, that there must be at least three relations between A and C. Finally, CVC could enforce that, if there exists an relationship between A and B, the number of embedding relationships in the model must be even.

These are all constraints that cannot be imposed by the DSL definition. Therefore, the model generator cannot take these constraints into account when generating models. If the model generator would be used to test such a DSL Application, we end up with virtually all generated models being invalid, and consequently with no usable test cases. Indeed, with each constraint that the CVC imposes on models, the model generator becomes less useful for creating test cases.

[24] and [30] provide an approach that could solve the problem stated above. In their approach for generating models from a metamodel defition, they define sub-domains and ranges for properties and multiplicities of relations in the metamodel. They define these sub-domains and ranges as *partitions*. A partition of a set S of elements is a division of S into one or more non-empty, non-overlapping subsets (ranges), so that the union of all subsets cover all the elements of S. For example, a partition of the set of Boolean values consists of the set of subsets $\{\{true\}, \{false\}\}$, where there are two ranges: a range that contains the value true and a range that contains the value false. In the papers, the partitions are used to define fragments, which represent combinations of partition ranges. Fragments are used for efficiently generating models so that as many parts of the metamodel are covered as possible.

Yet, the notion of partition ranges cannot completely solve the problem concerning CVC. Partition ranges are static, while CVC requires an dynamic approach for generating valid models, as illustrated in the problem above. To be able to create valid models, the behaviour of the model generator is customized by code that is referenced from the model generator at each step of the algoritm. This code is called the *custom hooks*. The custom hooks consist of methods that are called each time the model generator performs an action such as generating a new object or instantiating a property. The method has the ability to override the behaviour of the model generator by telling it what to do next. For example, at the

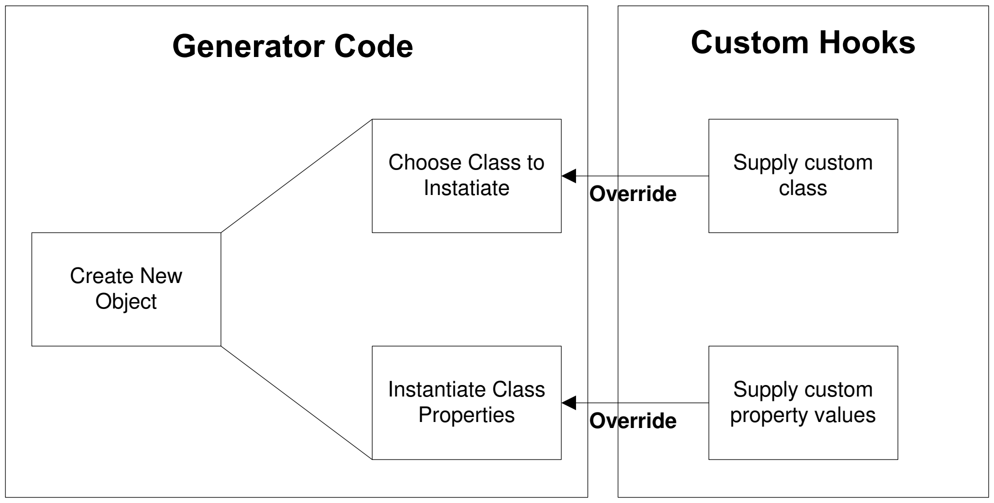

Generator Code
Custom Hooks
Choose Class to Instatiate
Supply custom class
Override
Create New Object
Instantiate Class Properties
Supply custom property values
Override

Figure 3.5: An example of how the custom hooks are connected to the model generator.

moment that the model generator chooses to instantiate a property of the a class, it invokes a method of the custom hooks. This method then can have code that identifies the property that is being instantiated and subsequently forces the model generator to instantiate the property as a certain value (e.g. a string that starts with "*each_name_should_start_with_this_string*").

Figure 3.5 illustrates how the custom hooks are connected to the model generator. The figure shows what happens when the generator creates a new object and how the custom hooks can alter the behaviour of the generator. When creating a new object, the generator performs two steps. First it randomly decides what subclass of the domain class it will generate. Then, after it has created a new instance of the chosen class, it instantiates all the properties of that instance by creating random values for each property (e.g. creating random strings or random integers). The custom hooks can alter the first step by supplying the model generator with a custom subclass that will be instantiated, instead of the subclass that the model generator chose. During the second step, the custom hooks can supply custom property values to the model generator, overriding any random property values that the generator had created. This way, the custom hooks make sure that the values of the properties fall within a valid range.

Some questions may arise as to the need of adding custom behaviour to the model generator. The example presented above does not seem very realistic. Yet, at the time that we used the model generator to create test cases for ACA.NET, and no custom behaviour was added to the generator, nearly all models were rejected by ACA.NET's CVC. From a set of 100 generated models, all of them were rejected by the CVC.

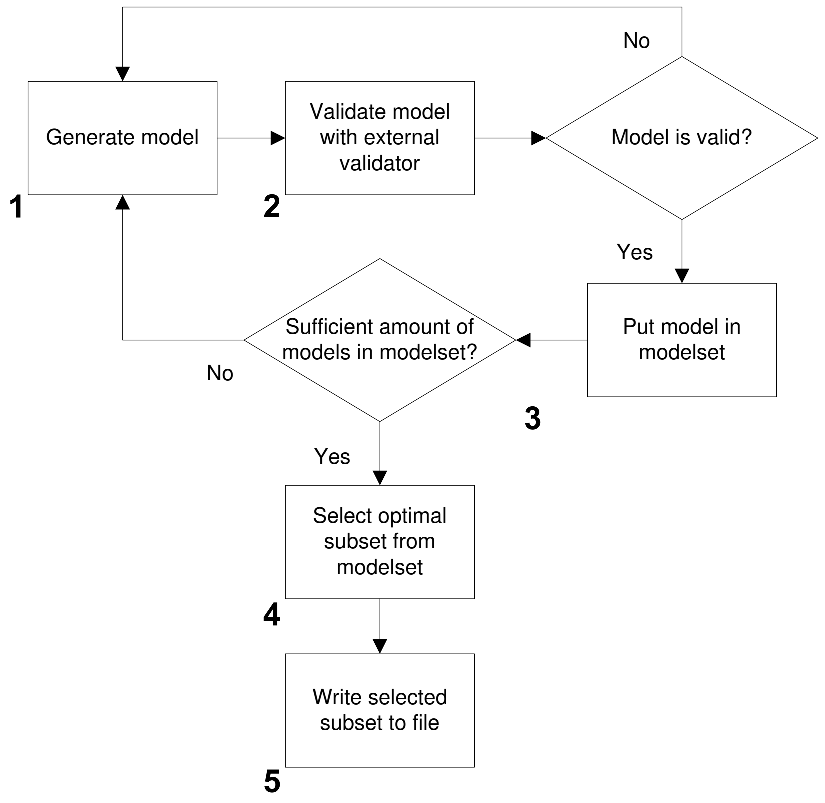
3.4.1 Implications of the Custom Hooks

Although the custom hooks do provide a solution to the restrictions that CVC impose of models, they do pose some problems as well. The necessity to add custom behaviour to the automatic model generator, so that it is able to generate valid models, actually makes the generator less generic. The idea of the generator is that it can generate valid models from any DSL definition. If custom code is required for generated valid models then, for each DSL, a developer should write a part of the model generator himself. That makes it less useful as a generic tool for testing DSLs. In addition, adding custom behaviour to the model generator imposes limits to the models that are generated. This makes it possible that some aspects of a metamodel are not properly tested. Using custom hooks can have some advantages as well. [30] explains that the effectiveness of partition testing strategies relies on the quality of the partitions that are used. If a tester does a good job in defining partitions, the generator will be able to efficiently generate models that tests the metamodel thoroughly.

3.5 Generating Model Sets

Generating one test case is usually not sufficient for testing a software system. The generator therefore has the ability to generate a set of test models. However, providing a DSL Application with large amounts of models, in order for it to generate code from it, takes up very much time. Depending on the DSL, supplying about one thousand models already takes hours to complete. It seems to be more useful to select a set of models from the generated models, so that many or all parts of a DSL are tested and only a small amount of models need to be provided to a DSL Application. This section explains how the generator creates sets of generated models. Figure 3.6 illustrates the steps for generating sets of models and subsequently choosing a subset and writing it to file. Here we describe these steps in more detail:

1. Generate a model.
2. We want each of the models to be valid so that it will be accepted by the DSL Application. The custom hooks influences the generation process in such a way that only valid models should be generated. Still, some constraints might not be handled by the custom hooks which results in some models being invalid. As we don't want to have invalid models in our test set, the model generator has gives an option to use an *external validator* to check if a given model is valid according to a DSL Application's CVC. An external validator is an instance of VS, which loads the generated model and validates it. When a model is generated, it is serialized to file and loaded into an instance of VS. The instance loads the file, recognizing that the file contains a model for a certain DSL Application. The model generator then tells the VS instance to validate the model. If any errors occur during the validation process, or the DSL Application marks the model as invalid, the model will not be added to the generated set of models. To make sure that there are enough models in the final model set, the model generator will make a new model for each model that has been rejected by the DSL Application (it returns to step 1).
3. If a model was found to be valid, it is added to the set of generated models. If the set of generated models is of sufficient size, we go to step 4, else we continue generating models by going to step 1.



No
Generate model
Validate model with external validator
Model is valid?
1
2
Yes
Sufficient amount of models in modelset?
Put model in modelset
No
3
Yes
Select optimal subset from modelset
4
Write selected subset to file
5

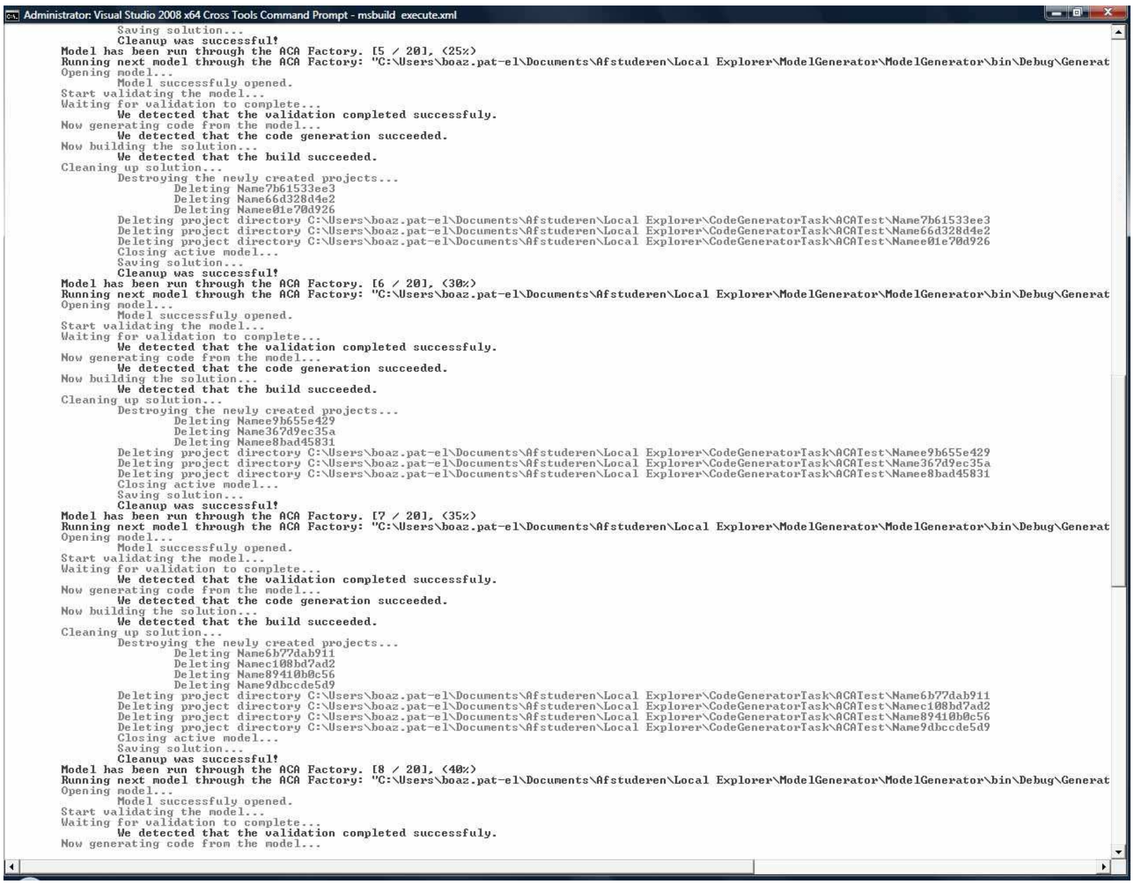
Figure 3.6: Algorithm for generating model sets.

4. Now that the generated set of models contains enough models, the generator chooses an optimal set from the set of generated models, according to certain coverage criteria, and marks these models as test set models. The test set is the set of models which will be used to test the code generator of the DSL Application. The process and justification for choosing an optimal subset from the set of generated models is explained in chapter 4.

5. Finally, the models in the testset are written to file, as is the data regarding the amount of metamodel coverage of the test set. Metamodel coverage is explained in chapter 4.

3.6 Code Generator Test Chain

Now that there is a set of test models for testing a DSL Application, it can be supplied with the generated test set. For this purpose, a *Code Generator Test Chain* (TC) was implemented. The purpose of the TC is to load a VS instance, supply it with a model from the set of test models, validate it, generate code from it, and finally compile the generated code. If any errors occur during these steps, the TC stops the process and writes to file the data


Administrator: Visual Studio 2008 x64 Cross Tools Command Prompt - msbuild execute.xml

```
                Saving solution...
                Cleanup was successful!
        Model has been run through the ACA Factory. [5 / 20], (25%)
        Running next model through the ACA Factory: "C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\ModelGenerator\ModelGenerator\bin\Debug\Generat
        Opening model...
                Model successfuly opened.
        Start validating the model...
        Waiting for validation to complete...
                We detected that the validation completed successfuly.
        Now generating code from the model...
                We detected that the code generation succeeded.
        Now building the solution...
                We detected that the build succeeded.
        Cleaning up solution...
                Destroying the newly created projects...
                        Deleting Name7b61533ee3
                        Deleting Name66d328d4e2
                        Deleting Namee01e70d926
                Deleting project directory C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\CodeGeneratorTask\ACATest\Name7b61533ee3
                Deleting project directory C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\CodeGeneratorTask\ACATest\Name66d328d4e2
                Deleting project directory C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\CodeGeneratorTask\ACATest\Namee01e70d926
                Closing active model...
                Saving solution...
                Cleanup was successful!
        Model has been run through the ACA Factory. [6 / 20], (30%)
        Running next model through the ACA Factory: "C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\ModelGenerator\ModelGenerator\bin\Debug\Generat
        Opening model...
                Model successfuly opened.
        Start validating the model...
        Waiting for validation to complete...
                We detected that the validation completed successfuly.
        Now generating code from the model...
                We detected that the code generation succeeded.
        Now building the solution...
                We detected that the build succeeded.
        Cleaning up solution...
                Destroying the newly created projects...
                        Deleting Namee9b655e429
                        Deleting Name367d9ec35a
                        Deleting Namee8bad45831
                Deleting project directory C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\CodeGeneratorTask\ACATest\Namee9b655e429
                Deleting project directory C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\CodeGeneratorTask\ACATest\Name367d9ec35a
                Deleting project directory C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\CodeGeneratorTask\ACATest\Namee8bad45831
                Closing active model...
                Saving solution...
                Cleanup was successful!
        Model has been run through the ACA Factory. [7 / 20], (35%)
        Running next model through the ACA Factory: "C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\ModelGenerator\ModelGenerator\bin\Debug\Generat
        Opening model...
                Model successfuly opened.
        Start validating the model...
        Waiting for validation to complete...
                We detected that the validation completed successfuly.
        Now generating code from the model...
                We detected that the code generation succeeded.
        Now building the solution...
                We detected that the build succeeded.
        Cleaning up solution...
                Destroying the newly created projects...
                        Deleting Name6b77dab911
                        Deleting Namec108bd7ad2
                        Deleting Name89410b0c56
                        Deleting Name9dbccde5d9
                Deleting project directory C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\CodeGeneratorTask\ACATest\Name6b77dab911
                Deleting project directory C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\CodeGeneratorTask\ACATest\Namec108bd7ad2
                Deleting project directory C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\CodeGeneratorTask\ACATest\Name89410b0c56
                Deleting project directory C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\CodeGeneratorTask\ACATest\Name9dbccde5d9
                Closing active model...
                Saving solution...
                Cleanup was successful!
        Model has been run through the ACA Factory. [8 / 20], (40%)
        Running next model through the ACA Factory: "C:\Users\boaz.pat-el\Documents\Afstuderen\Local Explorer\ModelGenerator\ModelGenerator\bin\Debug\Generat
        Opening model...
                Model successfuly opened.
        Start validating the model...
        Waiting for validation to complete...
                We detected that the validation completed successfuly.
        Now generating code from the model...
```

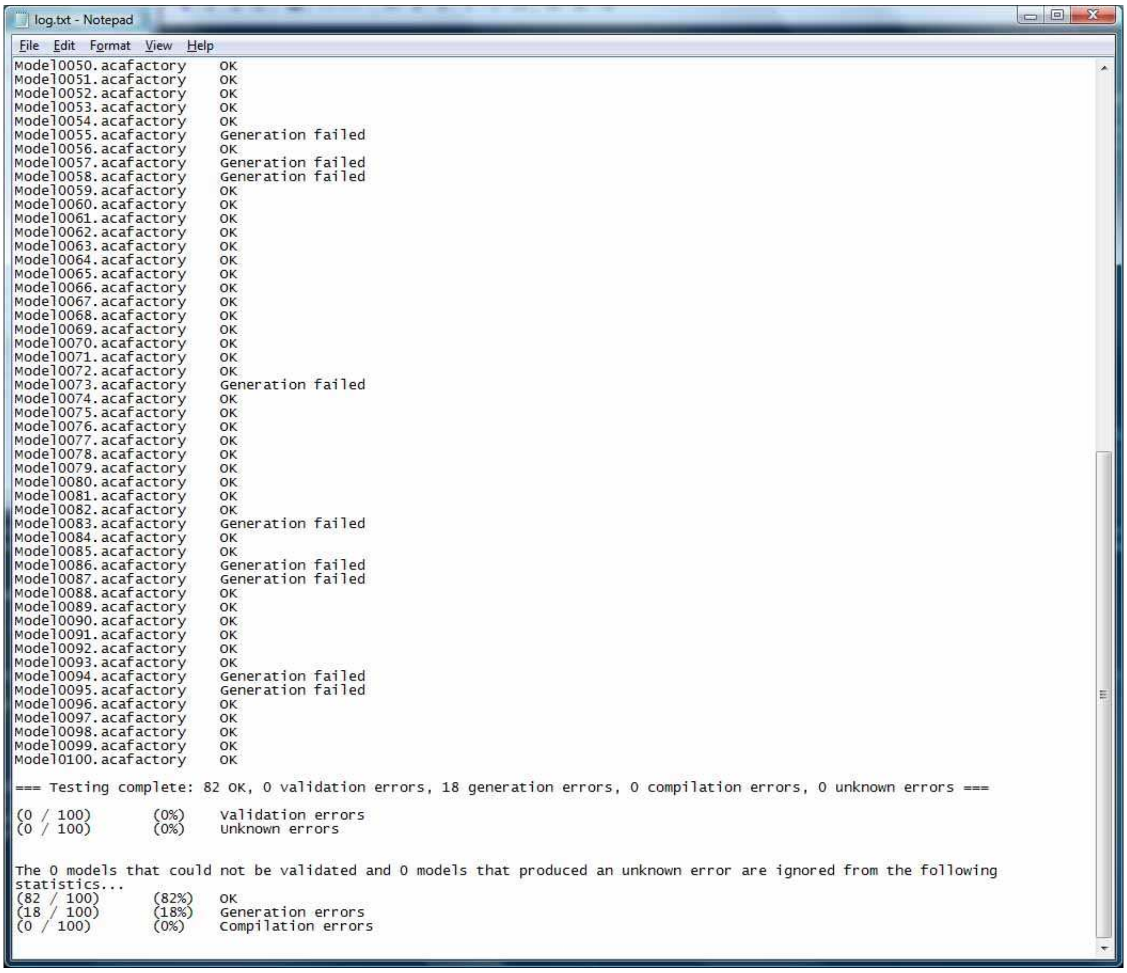
Figure 3.7: A running instance of the TC. In this case, 20 models were supplied to the DSL Application of ACA.NET.

related to the error. It then continues with the next model, until all models in the test set have been supplied to the TC. If an error occurs, data related to the error is written to file. The data related to an error are the following:

- When did the error occur? Was it during the validation, generation, compilation or any other step?
- What was the output of the DSL Application? All output (including exceptions and stacktraces that are outputted by the DSL Application) related to the error is written into a seperate file so that it can be inspected later. In addition, the filename of the log is the same as the name of the modelfile, so that any error logs can be easily related to a model.

Figure 3.7 shows a running instance of the TC.

After the test run has been completed, all data regarding the test run is summarized into a log-file. This log-file contains the following information:



```
Model0050.acafactory    OK
Model0051.acafactory    OK
Model0052.acafactory    OK
Model0053.acafactory    OK
Model0054.acafactory    OK
Model0055.acafactory    Generation failed
Model0056.acafactory    OK
Model0057.acafactory    Generation failed
Model0058.acafactory    Generation failed
Model0059.acafactory    OK
Model0060.acafactory    OK
Model0061.acafactory    OK
Model0062.acafactory    OK
Model0063.acafactory    OK
Model0064.acafactory    OK
Model0065.acafactory    OK
Model0066.acafactory    OK
Model0067.acafactory    OK
Model0068.acafactory    OK
Model0069.acafactory    OK
Model0070.acafactory    OK
Model0071.acafactory    OK
Model0072.acafactory    OK
Model0073.acafactory    Generation failed
Model0074.acafactory    OK
Model0075.acafactory    OK
Model0076.acafactory    OK
Model0077.acafactory    OK
Model0078.acafactory    OK
Model0079.acafactory    OK
Model0080.acafactory    OK
Model0081.acafactory    OK
Model0082.acafactory    OK
Model0083.acafactory    Generation failed
Model0084.acafactory    OK
Model0085.acafactory    OK
Model0086.acafactory    Generation failed
Model0087.acafactory    Generation failed
Model0088.acafactory    OK
Model0089.acafactory    OK
Model0090.acafactory    OK
Model0091.acafactory    OK
Model0092.acafactory    OK
Model0093.acafactory    OK
Model0094.acafactory    Generation failed
Model0095.acafactory    Generation failed
Model0096.acafactory    OK
Model0097.acafactory    OK
Model0098.acafactory    OK
Model0099.acafactory    OK
Model0100.acafactory    OK

=== Testing complete: 82 OK, 0 validation errors, 18 generation errors, 0 compilation errors, 0 unknown errors ===

(0 / 100)       (0%)    Validation errors
(0 / 100)       (0%)    Unknown errors

The 0 models that could not be validated and 0 models that produced an unknown error are ignored from the following
statistics...
(82 / 100)      (82%)   OK
(18 / 100)      (18%)   Generation errors
(0 / 100)       (0%)    Compilation errors
```

Figure 3.8: A log that has been produced by the code generator test chain.

- For each model, did it make it all the way through the TC without problems? Otherwise, indicate in which step an error occurred.

- What percentage of the models did make it through the validation step? Any models that did not make it through this step, or produced an unknown error that is not related to generation or compilation, are excluded from the follwing statistics.

- What percentage of the models did not produce an error, produced an error during code generation or produced an error during compilation?

Figure 3.8 shows an example of a log that has been produced by the TC.

With the model generator generating test sets and the TC supplying these test sets to the DSL Applications, it is possible to test code generators. The resulting log-files provide us with insight regarding the amount of errors in code generator and during which process these errors occurred. Still, we need to somehow quantify how good a code generator has been tested. The next chapter will explain how quality of test sets are measured.

Chapter 4

Coverage Analysis

With different kinds of testing methods, one might wonder how well a test set tests a program under test. If a particular test set finds many bugs in a program, does that imply that the test set is adequate, or just that the program itself contains a considerable amount of bugs? How many bugs can be considered as being many? Does the program size or complexity of the program require more tests in the test set? Also, one would like to quantify how well a test set tests a program.

4.1 Quantifying Test Quality

To approach this problem, tools have been developed that measure certain properties of a test set, like how many statements of the program under test are covered by the test cases. Zhu [59] presents a number of criteria on which quality test sets are measured. These criteria are based on code coverage: the amount of program statements that are executed when running the program with input test data.

The amount of coverage can be used as a quantity to indicate how well a test set performs [47]. Malaiya et al. [43] list some research on the relation between test coverage and the effectivess of finding errors in a program. He uses a mathematical model to review the relation between test coverage and fault coverage, which is the percentage of errors found regarding the estimated total amount of errors. He finds that an increase in test coverage (branch), significantly increases the amount of errors that are found.

These results indicate that the quality of a test set (how well a test set performs in finding errors) can be inferred from the amount of coverage of a test set. As the purpose of our research is to improve the testability of code generators built with DSL Tools, evaluating the quality of generated test sets should improve the quality of the overall testing process. With the improved quality of the testing process, the quality of the software system should increase as well.

4.1.1 White Box vs. Black Box Testing

The coverage criteria described above are *white box* criteria that assume knowledge of internal logic or code structure of a program is available. With the source code, the internal structure of a program can be verified to be working as intended. The tester derives test data from examining the program's logic and data flow [23, 30]. *Black box* criteria are not concerned with the internal structure of the program. Only the external behavior is evaluated to be equal to the program's specifications. Black box testing techniques are generally used for testing activities such as performance testing and robustness testing [23].

In our research, we have chosen for black box test adequacy criteria for quantifying the quality of test sets. This way, the approach to automatically generate test cases can be as generic as possible, so that it can be easily used in any SF development process. Choosing for black box criteria implies that code coverage criteria are not applicable to the approach of testing model transformations via the model generator, as code coverage criteria are white box criteria. However, the set of possible input models for a model transformator is completely specified by the input metamodel [30]. For that reason, we assess the quality of model sets by evaluating the coverage of the input metamodel. Note that the metamodel description (in the case of DSL Tools, a DSL Definition) is part of the system under test and, as such, our approach could be considered as a white box approach. Still, the code that makes up the code generator, the part of the system that is under test, is unknown and not considered for building test cases. Therefore, the approach is regarded as a black box approach.

The remainder of this chapter describes what metamodel coverage criteria have been used to quantify the quality of generated test sets. After that, the algorithms are explained that calculates how well a test set scores on coverage criteria. Finally, the process of choosing an optimal subset, regarding metamodel coverage, from the set of generated models is described.

4.2 Metamodel Coverage

In order to quantify the metamodel coverage of test models, the notion of metamodel coverage must be defined. In DSL Tools, a metamodel is a DSL description which consists of three elements: domain classes, domain relationships and domain properties. So, metamodel coverage can be defined as the amount of classes, relationships and properties of the DSL that have been instanced in the test models. Therefore, we propose that metamodel coverage can be determined by the following criteria: the amount of domain classes, the amount of domain relationships and the amount of domain properties that have been instanced in a model test set. Therefore, in order for a test set to achieve total metamodel coverage, it should instance all these elements that define the metamodel.

Now it would seem that all elements of a DSL have been addressed through these criteria. Yet, there are some implicit DSL elements that need to be taken into account for determining the amount of metamodel coverage. For example, if a domain class A has a relationship with a domain class B, then any subclasses of A can have an instance of the relationship as well. There are implicit relationships in the DSL definition between the subclasses of A and B. Similarly, domain properties are also inherited by subclasses of a domain class. [56] proposes the notion of *inheritance coverage* to account for these implicit relationships and properties. They define inheritance coverage between a superclass and a subclass as the coverage of the inherited features from the superclass in the subclass. If all inherited features from the superclass are covered in the subclass then full inheritance coverage is achieved for the subclass. Yet, the behaviour of the model generator has to be taken into account as well. Each time a domain class is instanced, all its properties (including its inherited ones) are instanced as well. Consequently, we can assume that, if a subclass has been instanced, all its inherited properties are instanced as well. Knowing how many inherited domain properties have been instanced can be derived from the number of subclasses that have been instanced. For this reason, in our definition of inheritance coverage, the number of inherited properties is not taken into account. It has already been taken into account in the class coverage measurements.

There are still DSL elements that have not been addressed by the proposed coverage criteria. A domain relationship in a DSL has source and target multiplicities. These multiplicities denote the type of constraints that are placed on the amount of relations that an object can have of a certain type. Thus, DSL definitions include constraints imposed on multiplicities of relations. Let us take a class Class_A with domain relationships Relation_B, of which Class_A is the source of Relation_B. If this relationship has source multiplicity $0..1$ then we would like to have test models that contain an instance of Class_A with no Relation_B relations and an instance of Class_A with one Relation_B relation. This way, the whole range of source multiplicities is tested, with the whole range being $\{\{0\},\{1\}\}$. [30] explains that the idea of specifying ranges for properties can also be used for defining ranges for multiplicities. For example, a relationship with the source multiplicity $0..*$ can be tested with the ranges $\{\{0\},\{1\},\{x \| x \geq 2\}\}$.

Finally, [56] proposes the definition of metamodel coverage to be the combined coverage of all other coverage criteria. Total metamodel coverage is achieved when all properties and relations, including inherited ones, are covered in a model set. In our research, when we use the term metamodel coverage, we refer to the combination of all other coverage criteria that are used for determining metamodel coverage. Thus, to achieve metamodel coverage, a test set should instance all classes, (inherited) relationships, (inherited) properties and multiplicity constraints.

4.2.1 Quantifying Metamodel Coverage

We propose the following criteria for determining metamodel coverage of a DSL. For any given coverage type (class, relation, property, inherited relation or multiplicity constraint), the amount of coverage is the amount of metamodel elements of the type that have been instanced in the model test set, divided by the total amount of elements of the type in the metamodel definition.

- Class coverage: The amount of domain classes that have been instanced in a model test set, divided by the amount of domain classes in the DSL. The procedure to obtain class coverage is straightforward. We traverse all the models in the model test set and observe what classes are instanced. Any abstract classes in the DSL are not counted as being part of the total number of classes in the DSL, as abstract classes cannot be instanced.

- Relation coverage: The amount of domain relationships that have been instanced in a model test set, divided by the amount of domain relationships in the DSL. The procedure to obtain relation coverage is similar to obtaining class coverage. The difference between relation coverage and class coverage is that the inherited relations must be recognized as being an instance of the superrelationship[1].

- Property coverage: The amount of domain properties that have been instanced in a model test set, divided by the amount of domain properties in the DSL. The method to obtain property coverage is similar to obtaining class coverage. Yet, just like with relation coverage, any instanced inherited domain property must be linked to the property of the superclass in which the property was defined.

- Inheritance coverage: The amount of inherited relationsships that have been instanced in a model test set, divided by the amount of inherited relationships in the DSL. The method for calculating inheritance coverage works as follows. To determine the number of inherited relationships in a DSL, all domain classes in the DSL are traversed and, for each relationship that a class has, is determined whether the relationships is an inherited one. If a relationship is an inherited one, it is added to the number of inherited relationships in a DSL. Then, for all models in the test set, all relations in a model are traversed and, for each relation, it is determined if it is an instanced inherited relationship.

- Relation Multiplicity Constraint coverage: Relation multiplicity constraint coverage ($C_{constraint}$) is the coverage of the domain relationship multiplicity ranges that can be tested in the model test set. For a domain relationship, the amount of ranges that can be tested in the test model is dependent on the type of the relationship. In DSL Tools, there are four multiplicity types: zero-to-one ($0..1$), one (1), zero-to-many ($0..*$) and one-to-many ($1..*$). Each of these multiplicities have a range that can be tested in a test set:

  1. Zero-to-one: Can be tested with the range consisting of two subsets: $\{\{0\}, \{1\}\}$.
  2. One: Can be tested with the range consisting of one subset: $\{1\}$.
  3. Zero-to-many: Can be tested with the range consisting of three subsets: $\{\{0\}, \{1\}, \{x \| x \geq 2\}\}$.
  4. One-to-many: Can be tested with the range consisting of two subsets: $\{\{1\}, \{x \| x \geq 2\}\}$.

  Note that a domain relationship has two multiplicity types, and that both the source and target multiplicity combinations can be tested. Also, inherited relationships are not taken into account for determining the relation multiplicity constraint coverage.

  Let D_m be the amount of multiplicity ranges that can be tested in a model test set and I_m be the multiplicity ranges that have been tested in a model test set. Then the

[1]A superrelationship is a relationship from which inherited relationships (subrelationships) derive their properties.

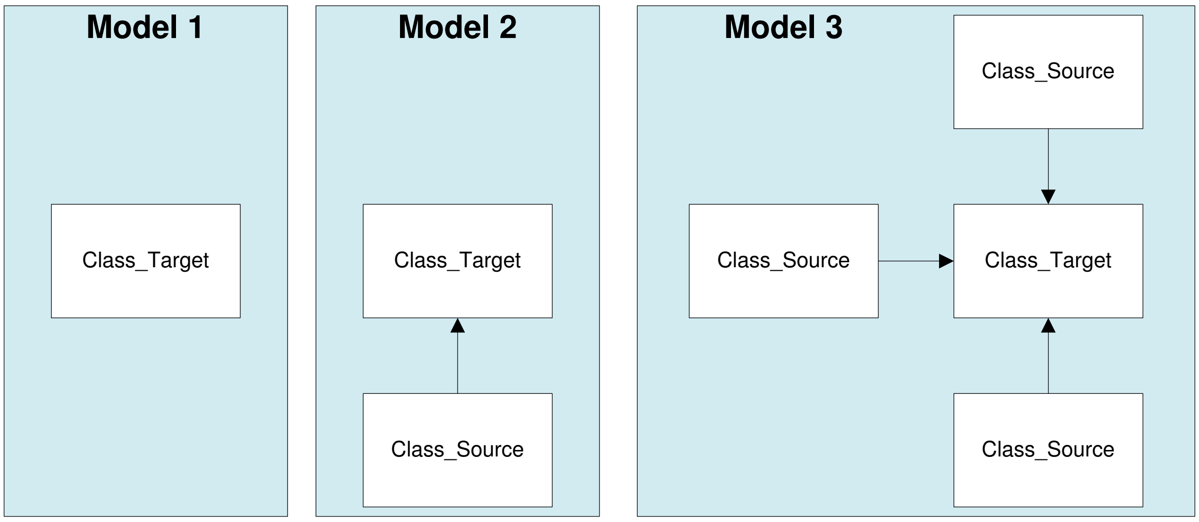


Figure 4.1: Set of three models that fully test the range of a relationship with source multiplicity 1 and target multiplicity $0..*$.

relation multiplicity constraint coverage can be calculated with

$$C_{constraint} = \frac{D_m}{I_m}$$

The method for calculating relation multiplicity constraint coverage consists of identifying the number of ranges that can be tested, and then determining what ranges have been tested in a model test set. The first step is to traverse the DSL and determine, for each relationship, what possible ranges can be tested. As an example, we take a relationship with source multiplicity 1 and target multiplicity $0..*$. Let S stand for source object, which is an object that is an instance of the source of the relationship) and T stand for target object, which is an object that is an instance of the target of the relationship. Then the relationship can be tested with the following models, illustrated in figure 4.1.

1. A model with at least one T, which does not have any instances of the relationship.
2. A model with a S with one instance of the relationship and a T with one instance of the relationship.
3. A model with a T with two or more instances of the relationship and two or more S's with one instance of the relationship.

This list corresponds to the source range $\{1\}$ and the target range $\{\{0\}, \{1\}, \{x \| x \geq 2\}\}$.

Then, all models in the model set are traversed and for each structure (subset of a range, like $\{1\}$) in D_m, it is determined whether it is tested in the model. If so, and the structure is not part of D_i, it is added to D_i.

- Metamodel coverage: The combined amount of class coverage, relation coverage, property coverage, inheritance coverage and relation multiplicity constraint coverage, divided by the amount of coverage criteria used to define metamodel coverage. Another metamodel coverage definition is possible: counting up all metamodel elements that have been instanced and dividing that amount by the total amount of metamodel elements. However, the amount of properties and relation multiplicity constraints is usually far greater than the number of classes and relations in a metamodel. In our point of view, property coverage is not more important than class or relation coverage. Therefore, the definition of metamodel coverage is the median of all other criteria.

The list above roughly describes how the criteria are calculated. In addition to the numbers, the software that was implemented to calculate the coverage also determines what classes, relations, constraints, etc., have not been covered by a model test set. For each generated test set, the data regarding all coverage criteria are written to file. The coverage data is calculated for each model as well. An illustration of such a file is presented in figure 4.2.

Set Coverage versus Model Coverage

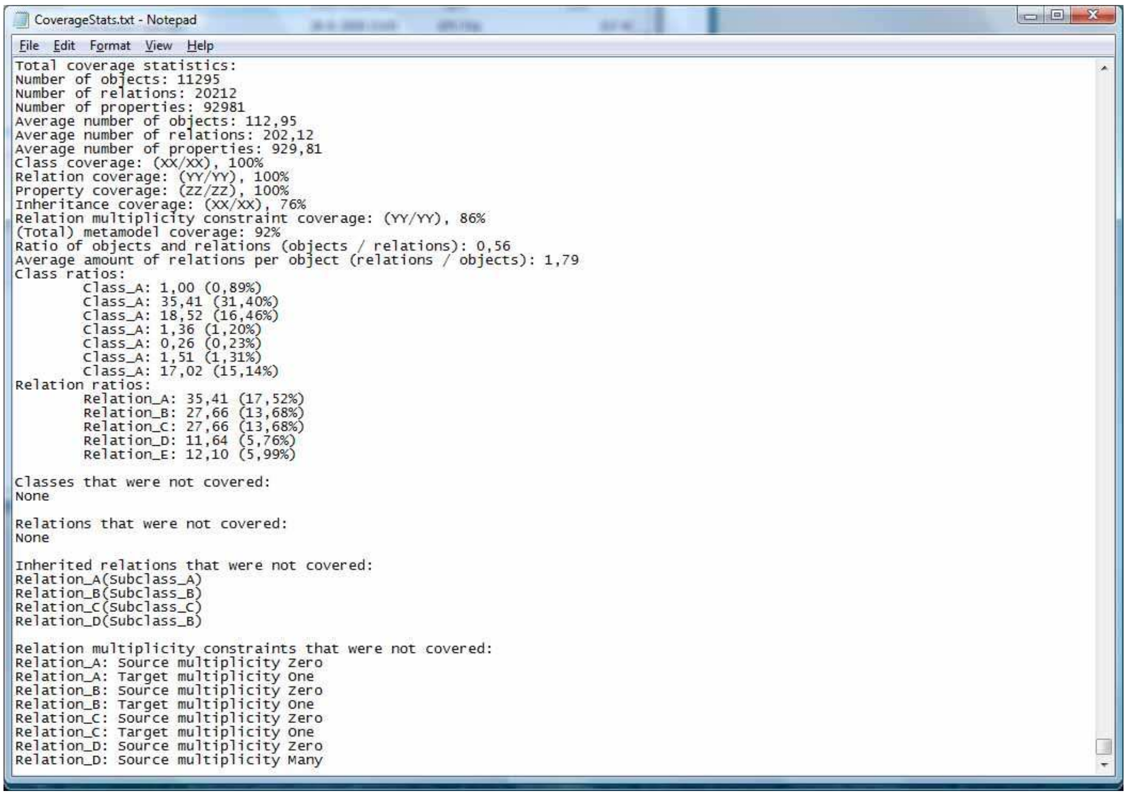
When generating test sets based on coverage criteria, we are faced with the following dilemma: should a test set have a high total coverage, or should a test set consists of models that individually have high coverage? For example, a test set consisting of thousands of models could have high metamodel coverage, although each model only covers a small part of the metamodel. A test set with lower coverage could consist of much less models, where each model has high metamodel coverage. So, should a tester choose for a high average coverage per model, or high total coverage per test set?

Naturally, there are both advantages and disadvantages to both choices. A set with many small models has the advantage that each test case is small, so the cause of any exposed error by a test model will, most probably, be more easily determined. On the other hand, the chance that a small model, with individually low metamodel coverage, will cause any errors to be exposed are smaller than with large models. Large models should have a better chance of exposing errors in model transformators and could find many different errors. However, the cause of the errors might be harder to find than with smaller models.

In our research, the choice was made to create both kinds of test sets. Thus, test sets were generated that consisted of small models and test sets were generated that consisted of large models that individually had high metamodel coverage. Chapter 5 shows a comparison of the results when using these different test sets for testing ACA.NET.

4.3 Structure Coverage

In coverage criteria for test sets, each individual test model covers some part of the metamodel. Combining all test cases results in sets that are able to cover large parts (or all parts)



```
Total coverage statistics:
Number of objects: 11295
Number of relations: 20212
Number of properties: 92981
Average number of objects: 112,95
Average number of relations: 202,12
Average number of properties: 929,81
Class coverage: (XX/XX), 100%
Relation coverage: (YY/YY), 100%
Property coverage: (ZZ/ZZ), 100%
Inheritance coverage: (XX/XX), 76%
Relation multiplicity constraint coverage: (YY/YY), 86%
(Total) metamodel coverage: 92%
Ratio of objects and relations (objects / relations): 0,56
Average amount of relations per object (relations / objects): 1,79
Class ratios:
        Class_A: 1,00 (0,89%)
        Class_A: 35,41 (31,40%)
        Class_A: 18,52 (16,46%)
        Class_A: 1,36 (1,20%)
        Class_A: 0,26 (0,23%)
        Class_A: 1,51 (1,31%)
        Class_A: 17,02 (15,14%)
Relation ratios:
        Relation_A: 35,41 (17,52%)
        Relation_B: 27,66 (13,68%)
        Relation_C: 27,66 (13,68%)
        Relation_D: 11,64 (5,76%)
        Relation_E: 12,10 (5,99%)

Classes that were not covered:
None

Relations that were not covered:
None

Inherited relations that were not covered:
Relation_A(Subclass_A)
Relation_B(Subclass_B)
Relation_C(Subclass_C)
Relation_D(Subclass_B)

Relation multiplicity constraints that were not covered:
Relation_A: Source multiplicity Zero
Relation_A: Target multiplicity One
Relation_B: Source multiplicity Zero
Relation_B: Target multiplicity One
Relation_C: Source multiplicity Zero
Relation_C: Target multiplicity One
Relation_D: Source multiplicity Zero
Relation_D: Source multiplicity Many
```

Figure 4.2: A file containing coverage data regarding a generated model test set for ACA.NET 5.1. Note that details regarding the ACA.NET metamodel have been removed.

of a metamodel. The proposed metamodel coverage criteria qualify the test cases by looking at what parts of the metamodel are instanced. This implies that a test set consisting of only one model could have the same metamodel coverage as a test set consisting of 10000 models. Does the test set with only one model perform better in exposing errors than the test set with 10000 models? In addition, two models of different sizes can have the same amount of metamodel coverage. Does the large model have a better chance of exposing an error? Although it does not have to be the case, we assume that models with complex structures generally have a better chance of exposing errors in model transformators than models without complex structures. For this reason, another coverage criterium is proposed based on the complexity of test models.

4.3.1 Model Complexity

In order to understand such a coverage criterium, it is necessary to define the complexity of a model. A model, an instance of a DSL Tools metamodel, can be considered as a connected rooted graph. Therefore, the complexity of a model can be regarded to be the same as the complexity of the graph that makes up its internal structure. Yet, determining the complexity of a graph is not a trivial matter. There is no universal concept of graph complexity and, as such, there are many different views on how it should be defined [38, 57, 48, 28]. In general, the complexity of a graph is based on the number of computing steps that is needed to solve some particular problem related to the graph, such as finding the shortest path between any two nodes.

4.3.2 Structure Coverage Criteria

The notions above do not take into account that the internal structure of a model depends on its metamodel. A DSL Definition is a graph made up of domain classes and domain relationships that determines what combinations of elements may occur in a model. Let a *structure* of a graph be a combination of connected elements in the graph. One metamodel allows for more or larger structures to exist than others. Hence, we propose a definition for model complexity, based on the amount of structures that are allowed to exist according to the metamodel definition. In other words, the complexity of a model is dependent on the number of combinations of connected elements that are allowed by the metamodel.

We illustrate structure coverage with an example. Let us consider a DSL Definition with two domain classes; Class_A and Class_B, and a relation R_{AB}, which is a reference relation between A and B and the relation R_{BB}, which is a reference relation between B and itself. Let A be an instance of Class_A and B be an instance of Class_B. To achieve full class coverage and relation coverage, a model only has to have a A with a relation R_{AB} to a B and a B with a relation R_{BB} to another B. Yet, the model does not have to contain the structure where a A has a relation to a B that has a relation to another B. Another structure that is not instanced is that of a B connected to another B which is connected to yet another B, while the DSL Definition does allow such structures to exist.

A DSL Definition is a graph made up of domain classes and domain relationships. With structure coverage, the metamodel is traversed and all possible combinations (structures) of classes and relations, of a certain length, are identified. Then, for each model in the test set, it is determined what possible structures are implemented by the model. The total length of a structure is bound by a value because, usually, the number of possible structures is infinite.

4.3.3 Alternative Complexity Criteria

Graph complexity is usually researched in the context of some problem related to graphs. The complexity of a graph provides insight into the number of computing steps that is needed to solve a problem regarding the graph. In search for complexity criteria for models, one could argue that building the model itself can be regarded as a measure for model complexity. Thus, the complexity of a model can be equal to the amount of steps needed to create the model. However, the amount of steps needed is primarily based on the amount of relations that are created during the model generation process. When creating a test set of models with a certain size, all these models would have roughly the same complexity, even though their underlying structure could be very different. Such a complexity measure is of no use as a coverage criterium.

Another notion of model complexity could be the amount of steps needed to generate code from the model. The steps needed to generate code from any model is determined by the model and the model transformation software. In the case of blackbox testing, the code of the code generation software is not known, and therefore the number of steps needed to transform a model into code is unknown. Hence, this method for defining model complexity is unsuitable as well.

4.4 Additional Measurements

How much does a generated model differ from a model that has been created by a human developer? Do generated models contain an excessive amount of objects or relations? These questions are not necessarily related to metamodel definitions and are not a part of the coverage measurements. Still, for our research, we wanted to have an impression of how generated models differ to human-made models. For that purpose, we measured the following properties of generated models:

- Average relations per object: The average amount of relations per object is equal to $\frac{NumberOfRelations}{NumberOfObjects}$.
- Average object depth: The depth of an object is the amount of embedding relations that connect the object to the root object. For example, the root object has depth 0, while an object directly embedded in the root has depth 1.
- Class ratio: For each class, the amount of instances of the class in a model, in relation to the amount of other objects in the model. So the class ratio for a specific class Class_A is $\frac{NumberOfClass\_AObjects}{NumberOfObjects}$.
- Relation ratio: Similar to class ratio, for each relation, the amount of instances of the relation in a model, in divided by the amount of relations in the model. So the relation ratio for a specific relation Relation_B is $\frac{NumberOfRelation\_BRelations}{NumberOfRelations}$.

4.5 Choosing an Optimal Test Set

When generating a set of test models for a DSL Application, the model generator starts by generating a set of models. We would like to make the model generator generate models in such a way that maximum metamodel coverage is achieved by choosing optimal generation strategies. Yet, forcing the model generator to generate models with certain properties makes the generator inflexible. Therefore, our approach involves generating a large amount of models and subsequently choosing a subset of models so that certain properties are satisfied. These properties are related to the coverage criteria described in the previous sections. For simplicity, we assume that we only want to maximize the class coverage of the test set. This means that we want our set of test models to instance as many classes in the DSL as possible.

We would like to select the mininum amount of models necessary to maximize the metamodel coverage of the test set. Each model in the set of generated models has one or more domain classes instanced. A model covering a domain class stands for the model having an instanced domain class in its description, so that each model covers a subset of all domain classes in the DSL. Figure 4.3 visualizes how models cover certain domain classes. Let S be the set of all covered domain classes in the generated set, and $C = \{S1, S2, ..., S_n\}$ a collection of subsets covered by the models in the generated set. The problem of choosing a minimum number of models so that as many classes in S are covered, is the problem of selecting as few as possible subsets from C such that their union covers S. This problem is the *set cover problem* and is NP-hard [54]. Accordingly, finding a polynomial-time algorithm is difficult or even impossible.

In our research, the number of models generated by the model generator exceeded ten thousand models. Consequently, using a brute-force algorithm for selecting an optimal test set proved to be too slow. Instead, a greedy algorithm was implemented to approximate the optimal set from the generated models. In a greedy algorithm, each step that is performed is locally the most optimal step. Let G be the generated set, T be the test set and m be a model. The pseudo-code below describes the algorithm. This algorithm determines the coverage for each $T \cup \{m\}$ in the generated set. The model m for which $T \cup \{m\}$ leads to the highest increase in coverage is added to T and removed from G. These steps are repeated until T is of sufficient size. The target size of T can be set by the user.

```
while <T.size < target_size> {
      highest_coverage := 0

      for each (m in G) {
            c := calculate_coverage(T + m)

            if (c > highest_coverage) {
                  model_with_highest_coverage := m
                  highest_coverage := c
            }
      }

      T.add(model_with_highest_coverage)
      G.remove(model_with_highest_coverage)
}
```

A disadvantage of using a greedy algorithm, like any other approximation technique, for solving the set cover problem is that it most probably only finds a local optimal solution to the problem, which is only a optimal solution among a certain set of solutions. In contrast, a global optimal solution is the best solution among all possible solutions. Yet, the greedy algorithm proves to be a good polynomial approximation algorithm for solving the set cover problem.

This algorithm creates a set of test models that can be used to test DSL Applications. In our research, these test sets where used to test two versions of ACA.NET. In the next chapter, we will describe how these tests were conducted and the results from those tests.

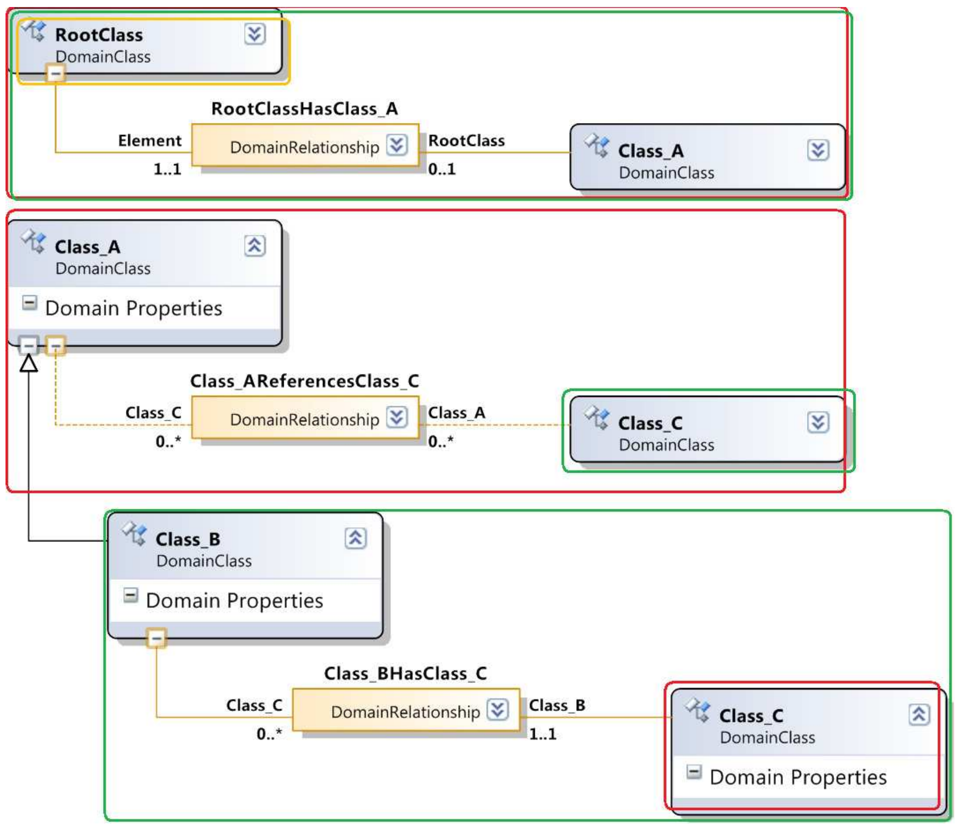


Figure 4.3: Each model instances one or more domain classes in a DSL. Each rectangle visualizes what classes from the DSL have been instanced by a model. Each different colour indicates a different model in the generated set.

Chapter 5

Experiments and Results

The goal of this thesis is to improve the testability of DSL Applications created using Microsoft DSL Tools. Our proposed approach for achieving better testability of DSL Applications is by using a tool that automatically generates test input for these applications. To understand whether such an approach actually improves the testability of DSL Applications, the following questions are to be answered. First, do we find errors in a DSL Application that were not exposed earlier? Any new errors that show up indicate that the testing tool for model transformations actually is an improvement over current manual testing. Second, do we find errors in a DSL Application that are still present in a later version of the application? If the tool is able to find errors in a DSL Application that are still present in a later version, then this suggests that the exposed errors are difficult to find using traditional testing approaches. Finally, in the previous chapter, several methods for qualifying test data were proposed. In order to determine whether these approaches are adequate for quantifying test quality, we need to know what relation exists between coverage criteria and their ability to predict the quality of test sets.

A case study has been performed in order to understand how the approach of automated test generation can improve the testability of DSL Applications. The approach for automated model generation has been tested on a ‘real-life’ subject; a business DSL Tools solution which has been used in multiple projects. Our case subject is ACA.NET 5.1, the second version of ACA.NET to use a DSL Tools language. There are two reasons for choosing a real-life subject instead of creating a self-made one to use as a test subject. First, the scope of ACA.NET is much bigger than that we are able to develop ourselves, given the time and people necessary to create a product of such size and scope. Secondly, ACA.NET has been tested quite thoroughly using unit testing, in addition to being used in multiple projects already. Because of this, it is possible to determine what additional value the approach of automated test generation contributes to the testing process, compared to the traditional testing methods that were used to test ACA.NET. The subsequent version of ACA.NET 5.1 is ACA.NET 6.0.

This chapter describes how the case study was setup, as well as the results from the study. In the following section, we explain what information is needed to answer the questions presented above. Then, the setup of the case study will be explained. The third section shows the results of performing test runs with different groups of test sets, that have different coverage properties. Finally, data is revealed regarding the performance of the model generator and Test Chain (TC).

5.1 Data to Acquire

In order to answer the questions posed in the previous section, the following must ascertained:

1. Do we find errors in ACA.NET 5.1 that were not exposed earlier?

    a) If any errors are found, how many errors are exposed in ACA.NET 5.1?

    b) What kind of errors are exposed in ACA.NET 5.1? Errors can be of the following types: validation, generation and compilation. If errors occur during the testing process, it will be determined at which stage they occur. This provides insight into what type of errors the model generator is able to find. The stage at which an error occurs can be inferred from the feedback from the TC.

    c) What is the source of the errors that are exposed? In other words, at which step lies the fault that resulted in an error? For example, if an error occurs during the code generation step, then that does not imply that there is an error in the generation code. Instead, there could be an error in the validation step which prohibited an invalid model to be filtered out, producing an error. The stage where the fault lies is determined by manually inspecting the model and source code related to the errors found in ACA.NET 5.1.

2. For each error found in ACA.NET 5.1, do we find these errors in ACA.NET 6.0? If an error was found in both versions of the ACAFactory, then this could provide some insight into what errors, exposed in ACA.NET 5.1, are particularly difficult to find by using traditional testing approaches.

3. What relation exists between coverage criteria and their ability to predict the quality of test sets?

    a) For different criteria, how many of the errors found in ACA.NET 5.1 are exposed by the test set? The coverage criteria that are considered in this thesis are metamodel coverage and structure coverage.

    b) What is the relation between a coverage criterium, the coverage value for a test set and the errors exposed in ACA.NET 5.1?

5.2 Experimental Setup

Multiple experiments have been run on ACA.NET using automatically generated test input. This section describes what tests have been created in order to test ACA.NET.

5.2.1 Input Test Models

The experiments are divided into two parts. In the first part, an attempt is made to find as many different errors as possible in ACA.NET using the model generator and indentifying the type (e.g. validation, generation, etc.) of the error and the source of the errors. The second part attempts to find a relation between the properties of a test set and the amount of different errors that can be found with a test set. This section describes the input test sets that have been constructed as part of the experiments.

5.2.2 Identifying Errors in the Code Generator

For the first part of the experiments, test models are needed that expose as many different errors in ACA.NET as possible, while it should still be possible to manually inspect the models as to determine the cause of the error. Although large models are probably able to find more errors than smaller ones, the first sets that were created consisted of small models. The first test set consisted of 10.000 models, with an average of 20 relations per model. The advantage of using relatively small models for testing the code generator is that identifying the cause of an error in a small model is easier.

Using this approach, a number of validation, generation and compilation errors were found in ACA.NET, and the causes were identified. Section 5.3.1 explains some of the errors that have been exposed.

Note that in our setup, one model can expose at most one error, even though it is possible for a model to have the potential to trigger multiple errors. Only the error that ACA.NET produces is taken into account. Usually, a tester would like to know if a model exposes more than one error. For instance, if an error that has been exposed by a model is fixed, does the model expose new errors that were hidden by the fixed error? Although we would like to investigate this option, it is beyond the scope of this thesis, as it involves fixing all exposed errors in ACA.NET and see if any new errors are found.

Finding Recurring Errors in ACA.NET 6.0

If any errors are found in ACA.NET 5.1, the question is whether any of these errors are also present in a subsequent version of ACA.NET. The next major version of ACA.NET 5.1 is ACA.NET 6.0. Therefore, tests are also run to find errors in ACA.NET 6.0. The test set for testing the code generator of ACA.NET 6.0 consisted of 2.000 models, with an average of 20 relations per model. As with testing the code generator of ACA.NET 5.1, the reason for choosing relatively small models as test input is that it is easier to identify the cause of an error small models. These 2000 models were chosen from a generated set of 10000 models and the selection was based on a model's average metamodel coverage. The reason for choosing average metamodel coverage as the criterium for test set quality is explained in sections 6.2.1 and 6.2.2. For any error that is exposed in ACA.NET 6.0, it is determined if the same error was also found while testing ACA.NET 5.1.

5.2.3 Determining Relation Between Coverage and Exposed Errors

Coverage values are used to quanitify test set quality. The higher the amount of coverage of a test set, the better we expect the set to be able to expose errors in a program. So, a test set with a high amount of coverage should perform better than a test set with a low amount of coverage, in terms of exposing errors in a program under test. Even if experiments show that test sets with high coverage consistenly expose more errors than test sets with low coverage, it is still unknown what the relation is between exposed errors and coverage. Thus, the question that needs to be answered is how the number of exposed errors is expected to increase as coverage increases.

In the second part of the experiments, an attempt is made to determine how the number of exposed errors changes with the amount of coverage. In order to determine this relation between coverage and exposed errors, there has to be data that can be used to derive such a relationship. A statistical method known as regression analysis has the ability to analyse such data. The goal of regression analysis is to determine the relationship between a independent variable (e.g. coverage) and a dependent variable (e.g. exposed errors). In this research, a nonlinear regression analysis has been performed. The reason for choosing nonlinear regression analysis instead of a linear regression analysis is that there is no reason to assume that the relation between coverage and exposed errors is strictly a linear one. Nonlinear regression analysis tests how multiple mathematical models fit on the experimental data. Then, a *measure of association* (or goodness of fit) analysis is performed to determine what model best fits the experimental data. The criterion for goodness of fit is the *proportion of explained variance*. Variance is the amount that the dependent variable (errors) changes in relation to the independent variable (coverage). The variance that can be explained by the coverage value is the proportion of explained variance. This method can give an indication that there is a relation between coverage and errors, and what kind of relation that would be. The nonlinear regression analysis has been performed using SPSS Statistics 17 [12], a predictive analytics software program.

Experimental data is needed to perform regression analyses. The experimental data consists of the number of errors found per coverage value. In this thesis, four different coverage criteria are analysed: total metamodel coverage, average metamodel coverage, total structure coverage and average structure coverage. Each of these criteria were analysed using a group of test sets. In each group of sets, all sets in the group differ on the coverage value of a particular coverage criterium. Thus, a group of sets that tests the relevance of total metamodel coverage, has test sets that have a total metmodel coverage value of 10%, 15%, 20%, etc. Then, for each group of sets, two additional groups of sets were created with the same properties as the first group. So, for each coverage value per criterium, three test runs were performed and all the resulting data is used for the regression analysis. The reason for performing multiple test runs is that each measurement contains random errors. Therefore, in order to reduce the error, three measurements are performed for each coverage value. The mean (average) of the three runs are used as for determining the relation between a coverage value and the amount of different errors found.

The values for coverage are not chosen arbitrarily. The lowest coverage in a group of sets is the lowest possible coverage that could be achieved by selecting 100 models from the

Coverage Method	Coverage Value Domains per Model Size		
	20	40	60
Total Metamodel	[10,00%, 75,00%]	[10,00%, 75,00%]	[10,00%, 75,00%]
Average Metamodel	[13,00%, 34,00%]	[25,10%, 48,10%]	[28,00%, 61,64%]
Total Structure	[5,10%, 53,50%]	[18,10%, 58,70%]	[10,80%, 60,30%]
Average Structure	[1,00%, 7,00%]	[5,50%, 13,00%]	[8,20%, 17,60%]

Table 5.1: Coverage values for test sets

generated set. The sets with the lowest coverage resemble the sets with the lowest predicted chance of exposing errors in the ACAFactory. Similarly, the highest coverage in a group of sets is the highest possible coverage that could be achieved by selecting 100 models from the generated set. The sets with the highest coverage resemble the sets with the highest predicted chance of exposing errors in the ACAFactory. Then, 8 coverage values between the lowest and highest possible coverage are tested.

The following groups of test sets were generated:

- Three groups of sets, where each set in a group has a different total metamodel coverage value. For models with 20 relations, the values range from 10% up to 75%. For models with 40 relations and 60 relations, value ranges are equal.

- Three groups of sets, where each set in a group has a different average metamodel coverage value. Average metamodel coverage denotes that each model in the test set has an average metamodel coverage of a certain value. For models with 20 relations, the values range from 10% to 75%. For models with 40 relations, the values range from 13% to 34%. For models with 60 relations, the values range from 28% to 54,6%.

- Three groups of sets, where each set in a group has a different total structure coverage value. For models with 20 relations, the values range from 5% to 50%. For models with 40 relations, the values range from 18,1% to 58,7%. For models with 60 relations, the values range from 10,8% to 60,3%.

- Three groups of sets, where each set in a group has a different average structure coverage value. Average structure coverage denotes that each model in the test set has an average structure coverage of a certain value. For models with 20 relations, the values range from 1% to 7%. For models with 40 relations, the values range from 5,5% to 13%. For models with 60 relations, the values range from 8,2% to 17,6%.

In each of the test sets, all models are of the same size. All test sets were created by generating 2.000 models and then choosing a subset of 100 models. Although it would seem better to have a larger generated set and test set, producing more than 2.000 models proved to be too time consuming due to the need to use the external validator. Performance is discussed further in section 5.4.

5.3 Results

This section lists the results of performing the test runs on the ACAFactory 5.1.

First, the results of finding errors in the ACAFactory 5.1 are shown. Then, the results of testing coverage criteria are described.

5.3.1 Exposing Errors in the ACAFactory 5.1

A total of 22.000 models were used to test the ACAFactory 5.1. Here we give an overview of the errors that were exposed, when these errors occurred and where the error originates from. A total of twelve errors were found in the ACAFactory 5.1. Eight of these errors resulted in exceptions being thrown during code generation, while four of these errors resulted in an exception being thrown during the compilation fase. Nine of the errors that were found in the ACAFactory were the result of inadequate validation of the input model. A model that actually was not supposed to be marked as valid was attempted to be transformed into code anyway, resulting in an error.

An example of such an invalid model is a model with a domain class that has a member of the Uniform Resource Identifier (URI) type. An URI can contain a port, which should be a positive integer number. Yet, a user can assign the URI-member a negative integer number. The ACAFactory marks the model as valid and starts the code generation process. This results in an error at the moment the URI is parsed. This is an example of an error that a user can reproduce using the ACAFactory editor. Yet, only two of the errors that were found using the generated models are errors that can be reproduced by using the ACAFactory editor. Other errors were caused by the transformation code. The following example will explain why some errors cannot be reproduced when using the visual designer to create models. The ACAFactory editor allows a developer to remove a certain shape from a model. The ACAFactory then removes any shapes and erases any members from shapes in the model, that are dependent on the removed shape. In this way, the ACAFactory prohibits the model from becoming invalid for transformation into code. However, with the model generator it is possible to create models with a structure that would not be possible to create by using the visual designer, while still being valid according to both the metamodel definition and the CVC. Therefore, the model generator is able to create models that are being marked as valid according to the ACAFactory, and subsequently exposing an error at the moment the model is transformed into code.

By far, most of the errors that are exposed in the ACAFactory cannot, or are very difficult, to reproduce using the visual designer. It is also interesting to note that Microsoft Visual Studio 2008$^{\text{TM}}$, the platform of the ACAFactory, crashed many times during the test runs. These errors occurred at the moment some models were opened. Yet, these errors could not be traced back to be caused by a particular model or an error in the ACAFactory, as these errors were not reproducable by reopening the same model that caused the error (? Verder uitleggen dat sequence of models ook niet de boel laat crashen, en dus lijkt totaal random).

5.3.2 Exposing Errors in the ACAFactory 6.0

A total of 2.000 models were used to find any errors in the ACAFactory 6.0 that were also found in the ACAFactory 5.1. With these tests, a total of 6 errors were exposed, of which 5 of those were also found in the ACAFactory 5.1. These errors occurred both during the generation step and the compilation step. Most of the errors were caused by the CVC of the ACAFactory being unable to mark all invalid models as being invalid. Also, the error that occurred because a user entered an invalid URI, used as an example in the section above, was one of the errors that was found to be present in both ACA.NET 5.1 and ACA.NET 6.0.

5.3.3 Testing Coverage Relevance

In the second part of the experiments, an attempt is made to find a relation between coverage criteria and their ability to predict the quality of test sets. The test sets for this part of the experiments are divided into two groups. The first group of test sets aims to find a relation between metamodel coverage and the amount of errors exposed. The second group of sets explores the relation between structure coverage and the number of errors found.

Figures 5.1 to 5.23 show the results of the testing the ACAFactory 5.1 with automatically generated test cases. There are two types of figures. The first one demonstrates the number of different errors that are exposed by the test cases. In these figures, the horizontal axis represents the (average or total) coverage value of the test set that was use for the measurement. The blue vertical bar represents the mean of the number of different errors that were found using three test sets with the same coverage value. The black line in the middle of the blue bar represents the error[1]. The mean number of different errors is calculated as follows.

Let d_i be the number of *different* errors that are found in the ACAFactory, when using test set i $(0 < i \leq 3)$ for testing the ACAFactory (all test sets have the same coverage value), then the mean $= \frac{d_1 + d_2 + d_3}{3}$.

The high and low values of the error bar are calculated as follows: $high = max(d_1, d_2, d_3)$; $Low = min(d_1, d_2, d_3)$.

The second type of figures demonstrates the results of the regression analysis and are composed of dots and lines. The dots resemble the measurements, while the lines are proposed models for the data. The red line resembles the model that best fits the data according to the proportion of explained variance. The higher the proportion of explained variance, the better the model fits the data.

Accompanied by each figure that shows the results of the regression analysis, is a table that gives details about the results of the regression analysis. This table is divided into two parts: one that describes fitness of the model and another that gives estimates for the coefficients for each parameter in the model. The fitness of the model is portrayed as R^2, which is the percent of variance in the dependent variable that can be explained by the model.

[1]Error in the reported measurement; The difference between the highest and lowest measurement compared to the mean.

	Model Fit	Parameter Estimates			
Equation	R^2	Constant	b1	b2	b3
Linear	,450	-,391	,030		
Logarithmic	,415	-2,639	,986		
Inverse	,316	1,608	-20,696		
Quadratic	,450	-,457	,034	-4,811E-5	
Cubic	,455	-,032	-,010	,001	-9,593E-6

Table 5.2: Summary of model fit and parameter estimates, total metamodel coverage, 20 relations per model

Let S_r be the explained (regression) sum of squares and S_e be the sum of squared errors (unexplained sum of squares). Then the total sum of squares $S_t = S_r + S_e$. R^2 can then be calculated as R$^2 = 1 - \frac{S_e}{S_t}$. As can be seen from the formula, the value of R^2 ranges from 0 to 1. The higher the value, the better the model fits the data.

The tables also show parameter estimates for the models that were fit on the data. For example, let *y* be the dependent variable and *x* be the independent variable. A linear model with parameter estimates: constant = 0,5 and b1 = 0,3 denotes that the line $y = 0,5 + 0,3x$, was used to fit the data.

To further explore how coverage relates to the number of errors, an investigation was done to see whether certain peaks in the number of exposed errors could be linked to sets having high (average) metamodel coverage or (average) structure coverage. For example, if sets with 10,29% structure coverage caused a (local) peak in the number of exposed errors, then it was verified whether these sets have high (average) metamodel coverage. From our findings, peaks in exposed errors do not appear to be related to high amounts of metamodel or structure coverage.

Total Metamodel Coverage

Figure 5.1 shows the results of performing test runs with different total metamodel coverage values and a relation count of 20 per model. In the first sets, with low coverage, no errors were exposed at all. Then, the number of errors increases slowly, with the maximum errors found reaching its maximum around 46,11%. Then the number of errors found remains steady, although the error slowly decreases, with the smallest error at 75% coverage.

In figure 5.2 the results of regression analysis on the acquired data are shown. It shows that the cubic model has the best fit on the data. The quadratic model is almost equal to the linear model and they only fit less well to the data as the quadratic model. Table 5.2 gives an overview of the models that have been tested on the data and explains how well they fit.

The R^2 values show that no particular model fits the data well. Even the model that fits best only has an an explained variance of 45%. Although sets with higher coverage values expose more errors, from this data it looks like there is no particular relation between total metamodel coverage and errors.

Figure 5.3 shows the results of the test runs with different total metamodel coverage

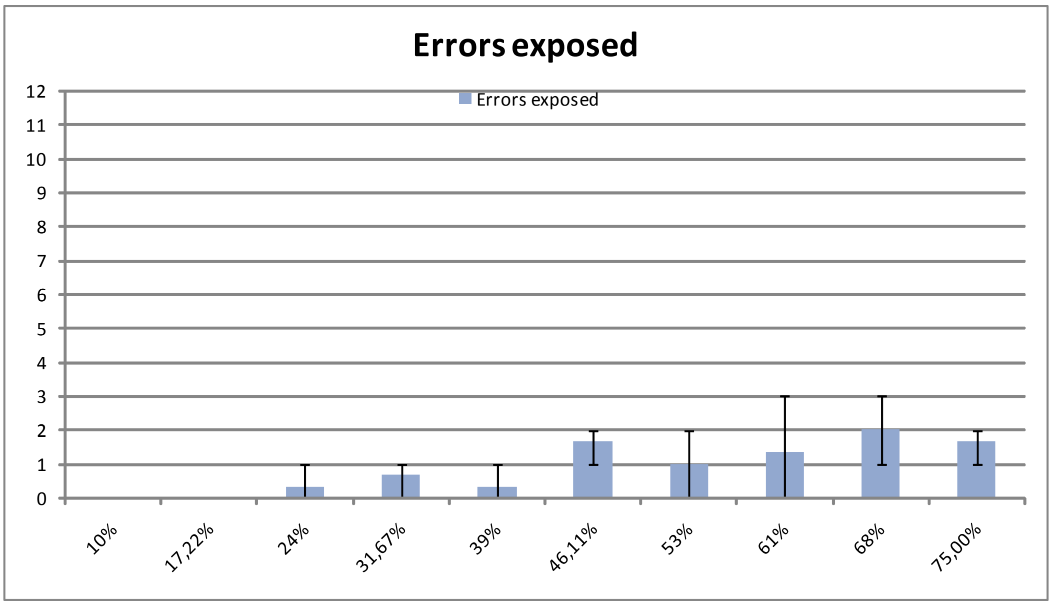


Figure 5.1: Errors exposed by sets with an average size of 20 relations with different total metamodel coverage.

values and a relation count of 40 per model. The number of errors found is low at the left side of the graph, increasing a little until 46,11% and 53%, where the number of errors found increases greatly. For higher coverage values, the number of errors increases, but not as sharply as between 46,11% and 53%.

Table 5.3 shows results that quite differ from the previous results. The R^2 values for linear, quadratic and cubic models seem to fit better on the data for models with 40 relations. The cubic model has the best fit on the data and the quadratic model does not fall far behind. Yet, when looking at figure 5.4 these two models demonstrate quite a different curve. The cubic model presents a relation where the number of exposed errors does not increase at all when metamodel coverage is increased, until a certain coverage value is reached. At that point, the errors found increases almost linearly with the increase in coverage. Then, at some point, no more errors will be exposed with increased coverage. According to the quadratic model, the number of exposed errors increases at the moment that metamodel coverage increases, but this increase in exposed errors slows down quickly.

In figure 5.5, the results are shown of the test runs with different total metamodel coverage values and a relation count of 60 per model. Between the highest and lowest coverage values, the number of errors found appears to be quite random, and the error also remains relatively high throughout all the results.

Indeed, table 5.4 illustrates that no model seems able to explain more than around 38% of the results. The cubic model performs best, even though it only performs slightly better than the linear model, which does not perform well. The results suggest no particular relationship could be found to exist between total metamodel coverage and errors, for test sets

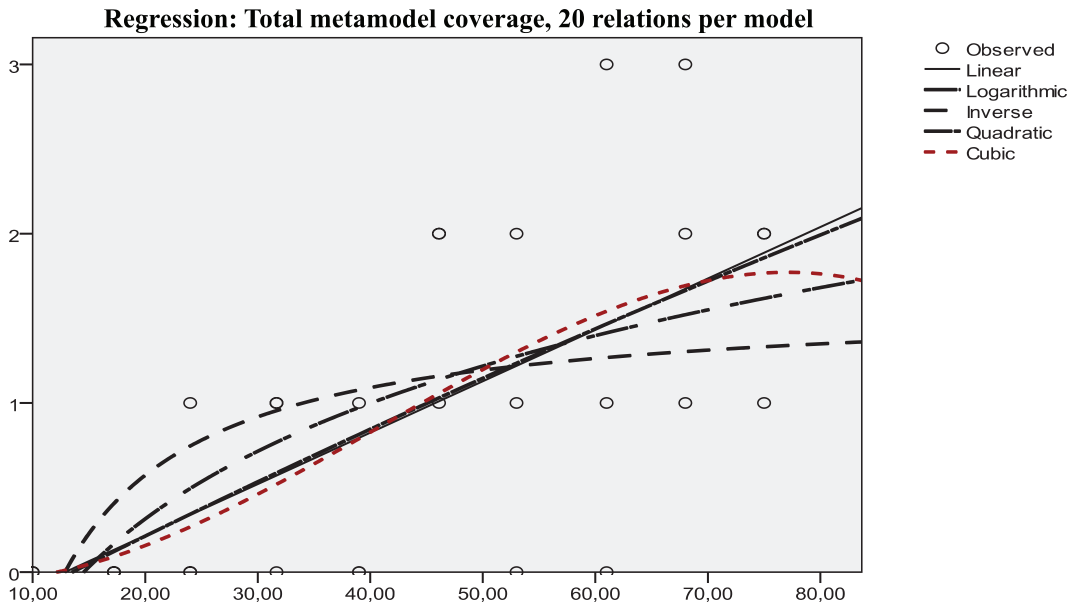


Figure 5.2: Visualisation of model estimates: Total metamodel coverage, 20 relations per model

	Model Fit	Parameter Estimates			
Equation	R^2	Constant	b1	b2	b3
Linear	,683	-,541	,063		
Logarithmic	,558	-4,759	1,920		
Inverse	,370	3,422	-37,676		
Quadratic	,707	,352	,008	,001	
Cubic	,726	1,831	-,148	,005	-3,339E-5

Table 5.3: Summary of model fit and parameter estimates, total metamodel coverage, 40 relations per model

containing models with 60 relations per model.

Figure 5.6 illustrates the models that have been tested on the results. Here we see that almost all curves appear linear, except for the quadratic and cubic models, which are almost equal. Still, it seems that the average number of exposed errors rises slightly as the coverage increases.

Average Metamodel Coverage

Figure 5.7 show the results of performing test runs with different average metamodel coverage and a relation count of 20 per model. In the figure, it can be seen that up until an average coverage of 15,33%, no errors are exposed. Then the amount of exposed errors

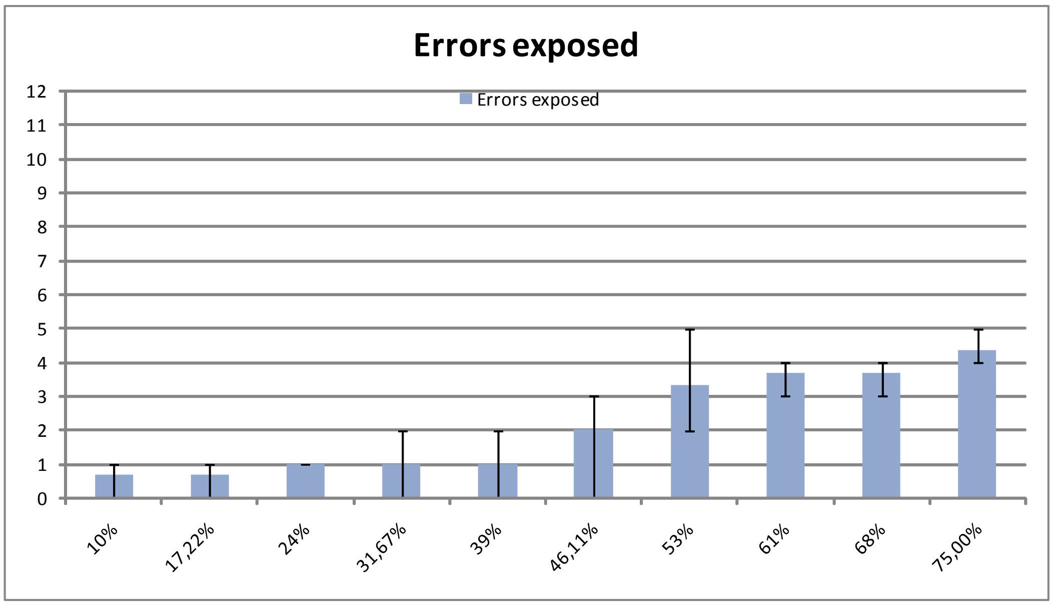


Figure 5.3: Errors exposed by sets with an average size of 40 relations with different total metamodel coverage.

	Model Fit	Parameter Estimates			
Equation	R^2	Constant	b1	b2	b3
Linear	,359	-,566	,065		
Logarithmic	,339	-10,220	3,327		
Inverse	,314	6,104	-160,045		
Quadratic	,376	2,916	-,070	,001	
Cubic	,377	1,833	,000	,000	9,258E-6

Table 5.4: Summary of model fit and parameter estimates, total metamodel coverage, 60 relations per model

rises sharply with a peak at 22,33%. Except for a single decrease at 24,66%, the number of errors found rise slighty with the highest amount reached at 31,66%. Although it seems that the number of exposed errors do not change from 27%, the error does become smaller.

Table 5.5 demonstrates the results of regression analysis on the acquired data. From the data, it seems that all models fit well on the data, with the cubic model fitting best. However, figure 5.8 presents a situation where all models are almost the same, following a nearly linear pattern.

In figure 5.9 the results are illustrated of performing test runs with different average metamodel coverage and a relation count of 40 per model. From the minimum coverage level, up to 35,32%, the number of errors found seem to increase linearly. Then the number

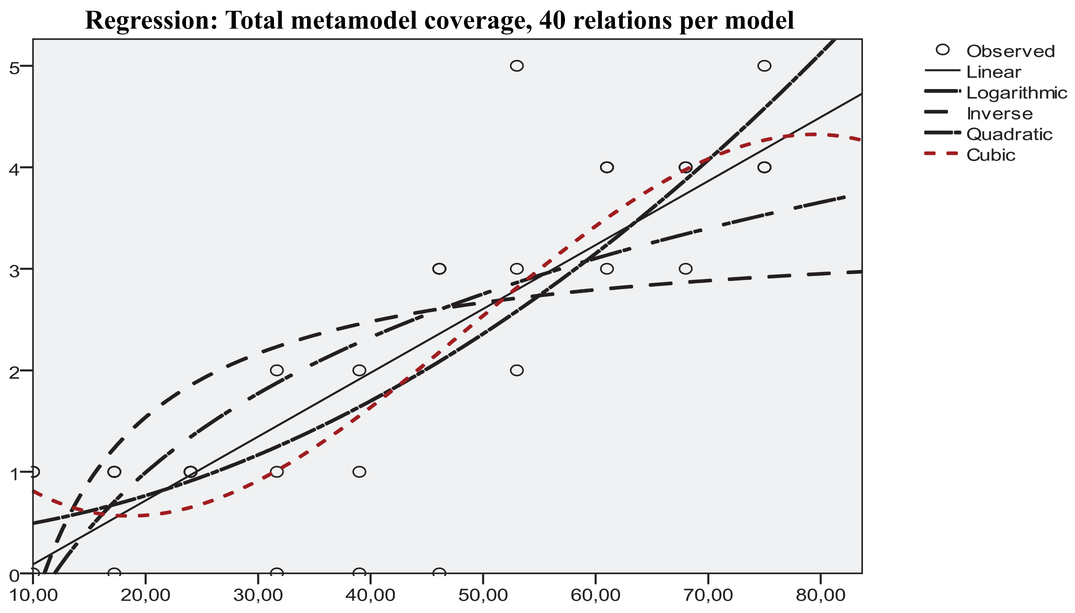


Figure 5.4: Visualisation of model estimates: Total metamodel coverage, 40 relations per model

	Model Fit	Parameter Estimates			
Equation	R^2	Constant	b1	b2	b3
Linear	,740	-2,672	,213		
Logarithmic	,770	-12,618	4,803		
Inverse	,767	6,917	-98,306		
Quadratic	,781	-7,029	,617	-,009	
Cubic	,783	-5,710	,427	,000	,000

Table 5.5: Summary of model fit and parameter estimates, average metamodel coverage, 20 relations per model

of exposed errors actually decrease a little, only to rise to a higher level at 45,55% with the highest number of errors found at the highest coverage level.

Table 5.6 gives the results of the regression analysis performed on the data. In this case, the inverse model fits best, but has only a marginally (0,1%) better R^2 value than the quadratic model. The cubic model is the same as the quadratic model. Just like with average metamodel coverage for models with a relation count of 20, figure 5.10 presents a situation where most models seem to follow an almost linear curve.

In figure 5.11 the results are shown of performing test runs with different average metamodel coverage and a relation count of 60 per model. From the figure, it appears that the average number of exposed errors rises almost consistently as the coverage increases, except for two decreases errors at 33,91% and 45,73%. The average number of exposed errors

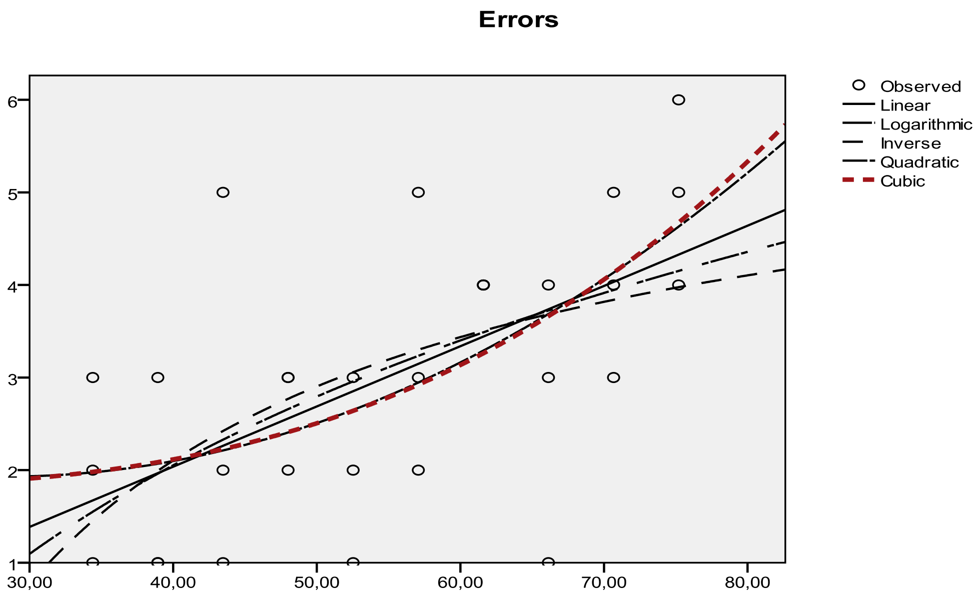


Figure 5.5: Errors exposed by sets with an average size of 60 relations with different total metamodel coverage.

	Model Fit	Parameter Estimates			
Equation	R^2	Constant	b1	b2	b3
Linear	,704	-2,498	,174		
Logarithmic	,737	-18,776	6,326		
Inverse	,760	10,161	-220,825		
Quadratic	,759	-12,200	,726	-,008	
Cubic	,759	-12,200	,726	-,008	,000

Table 5.6: Summary of model fit and parameter estimates, average metamodel coverage, 40 relations per model

at the highest coverage is about 2 to 3 times higher than at the lowest coverage, but the error remains relatively for all coverage values.

Table 5.7 reveals that it is difficult to find a model that explains the acquired data. There are three models that best fit the data: the logarithmic model, the quadratic model and the cubic model. Yet, these models only perform slightly better than the linear model and they are only able to explain 46% of the data. In fact, all models follow almost a linear curve, as can be seen from figure 5.12.

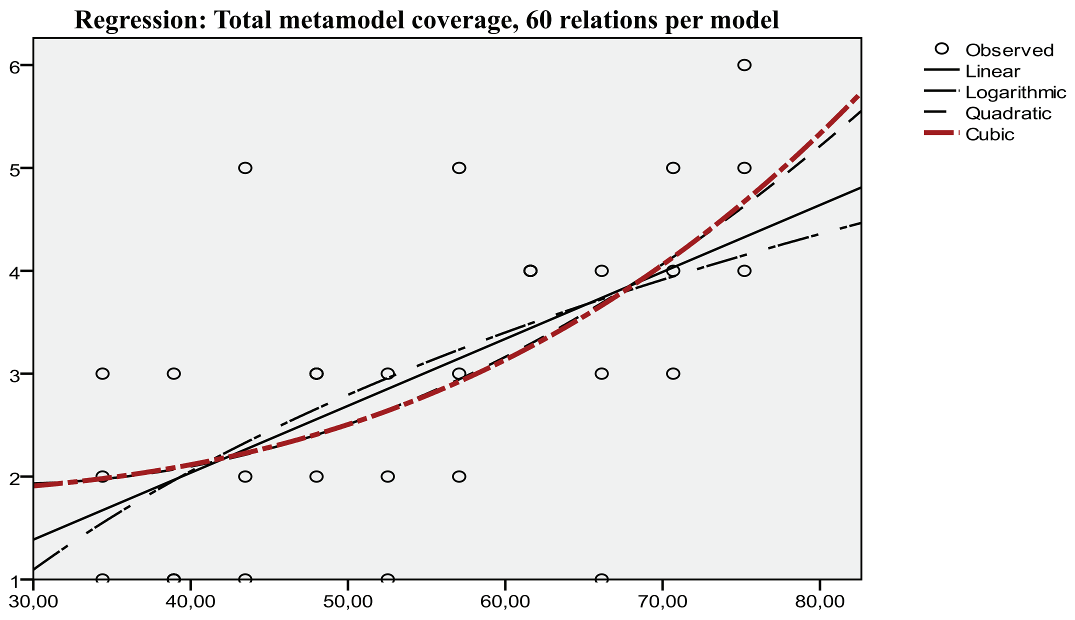


Figure 5.6: Visualisation of model estimates: Total metamodel coverage, 60 relations per model

Equation	Model Fit	Parameter Estimates			
	R^2	Constant	b1	b2	b3
Linear	,457	-,749	,132		
Logarithmic	,461	-14,926	5,306		
Inverse	,456	9,864	-203,952		
Quadratic	,461	-3,350	,263	-,002	
Cubic	,461	-2,520	,200	,000	-1,290E-5

Table 5.7: Summary of model fit and parameter estimates, average metamodel coverage, 60 relations per model

5.3.4 Testing Structure Coverage Relevance

Following the same approach for testing the relevance of metamodel coverage, this section shows the results of performing test runs on ACA.NET 5.1 with different structure coverage.

Total Structure Coverage

Figure 5.13 shows the results of performing test runs with different total structure coverage and a relation count of 20 per model. From the graph, it can be seen that the number of exposed errors increases for increased coverage. The sets with the smallest coverage reveal no errors, while test sets with increasing coverage exposed increasingly more errors. The increase in the number of exposed errors rises at a coverage between 26,61% and 48,12%,

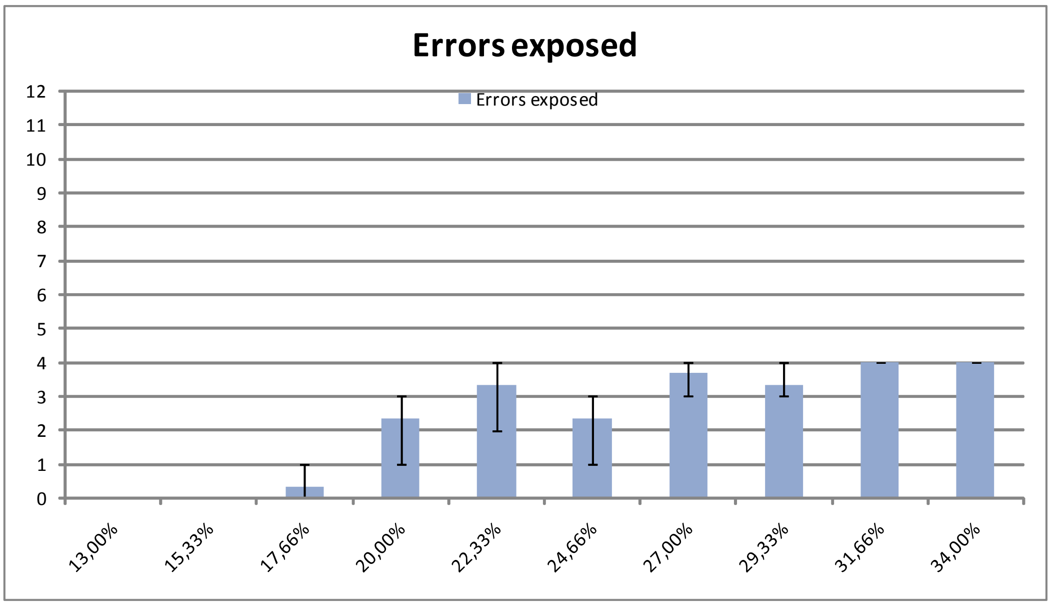


Figure 5.7: Errors exposed by sets with an average size of 20 relations with different average metamodel coverage.

Equation	Model Fit	Parameter Estimates			
	R^2	Constant	b1	b2	b3
Linear	,686	-,822	,080		
Logarithmic	,521	-3,319	1,527		
Inverse	,316	2,446	-16,291		
Quadratic	,752	,315	-,027	,002	
Cubic	,755	-,067	,037	-,001	2,996E-5

Table 5.8: Summary of model fit and parameter estimates, total structure coverage, 20 relations per model

and then decreases between 48,12% and 53,50%. The total number of exposed errors is 4. Note that up to a coverage level of 42,74%, the error is quite large.

According to 5.8, a quadratic or cubic model fits the model well, while figure 5.14 shows that these models almost follow the same curve through the data. A linear model performs somewhat less in contrast to the higher order models.

In figure 5.15 the results are shown of performing test runs with different total structure coverage and a relation count of 40 per model. In the graph, the number of exposed errors seems to be quite random in relation to the coverage value, although almost all of the test sets with a coverage of over 40,66% had a better performance in finding errors than those below 40,66%.

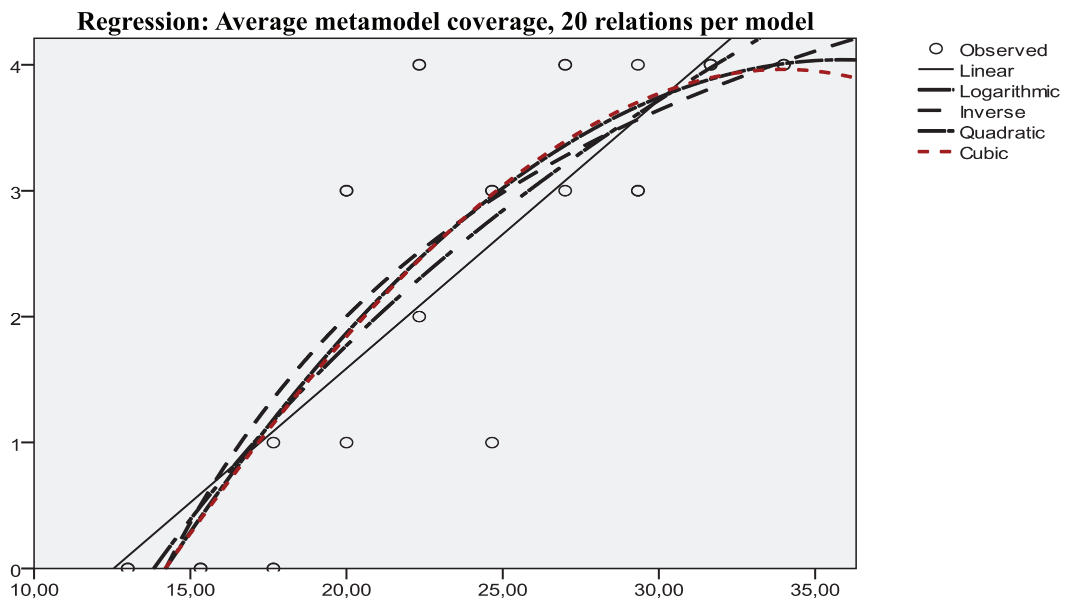


Figure 5.8: Visualisation of model estimates: Average metamodel coverage, 20 relations per model

	Model Fit	Parameter Estimates			
Equation	R^2	Constant	b1	b2	b3
Linear	,317	1,594	,062		
Logarithmic	,350	-4,195	2,276		
Inverse	,366	6,141	-73,073		
Quadratic	,367	-1,242	,229	-,002	
Cubic	,367	-1,051	,211	-,002	-4,230E-6

Table 5.9: Summary of model fit and parameter estimates, total structure coverage, 40 relations per model

Table 5.9 gives results indicating that no model, that has been tested using the regression analysis, fits well on the data. All models in 5.16 do show a curve that increases as the coverage increases. Yet, the results seem to indicate that there is no particular relation between total structure coverage and errors, for models with a relation count of 40.

Just like with sets consisting of smaller models, the results presented in figure 5.17 indicate that the relation between coverage and the exposed errors is somewhat vague. An average of 3 errors are found at the lowest coverage, while 5,5 errors are found at the highest coverage. Yet, the number of errors actually decreases consistently from 10,8% coverage to 32,8%. Then the errors inconsistently rises up.

As might be expected from such results, table 5.10 illustrate that it is difficult to fit a

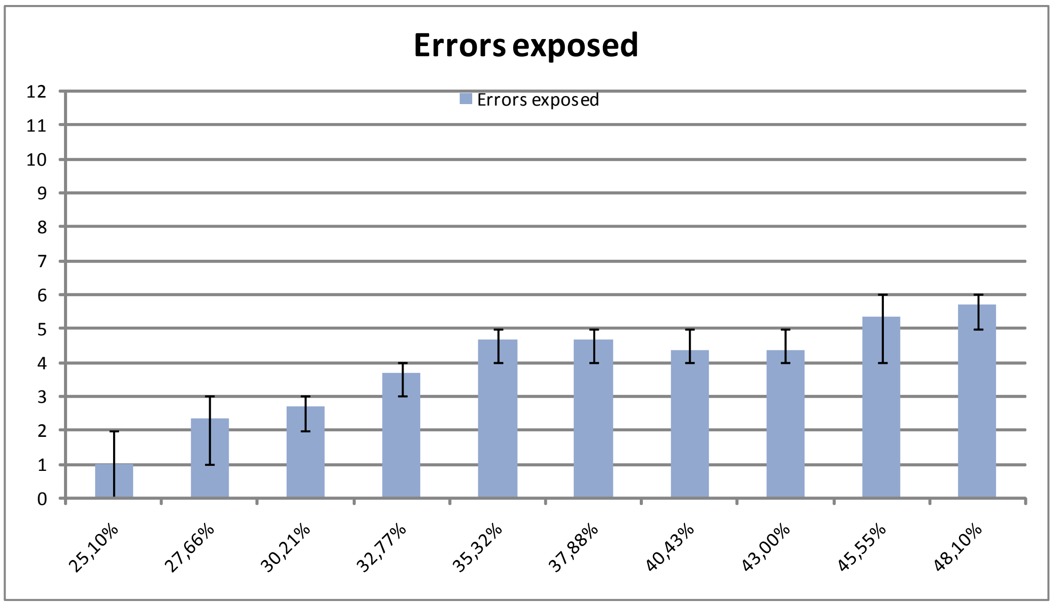


Figure 5.9: Errors exposed by sets with an average size of 40 relations with different average metamodel coverage.

	Model Fit	Parameter Estimates			
Equation	R^2	Constant	b1	b2	b3
Linear	,457	-,749	,132		
Logarithmic	,461	-14,926	5,306		
Inverse	,456	9,864	-203,952		
Quadratic	,461	-3,350	,263	-,002	
Cubic	,461	-2,520	,200	,000	-1,290E-5

Table 5.10: Summary of model fit and parameter estimates, total structure coverage, 60 relations per model

model on the acquired data. The models that best fit the data, the quadratic and the cubic models, only can explain around 46% of the data, which is only slightly better than the explained variance of 45,7% by the linear model. Still, figure 5.18 shows that the cubic model is quite different from the linear model. However, the curve shown in the figure also suggests a model where, at two points, the number of exposed errors actually decreases as the coverage increases, which does not seem like a realistic model for the relation between coverage and errors.

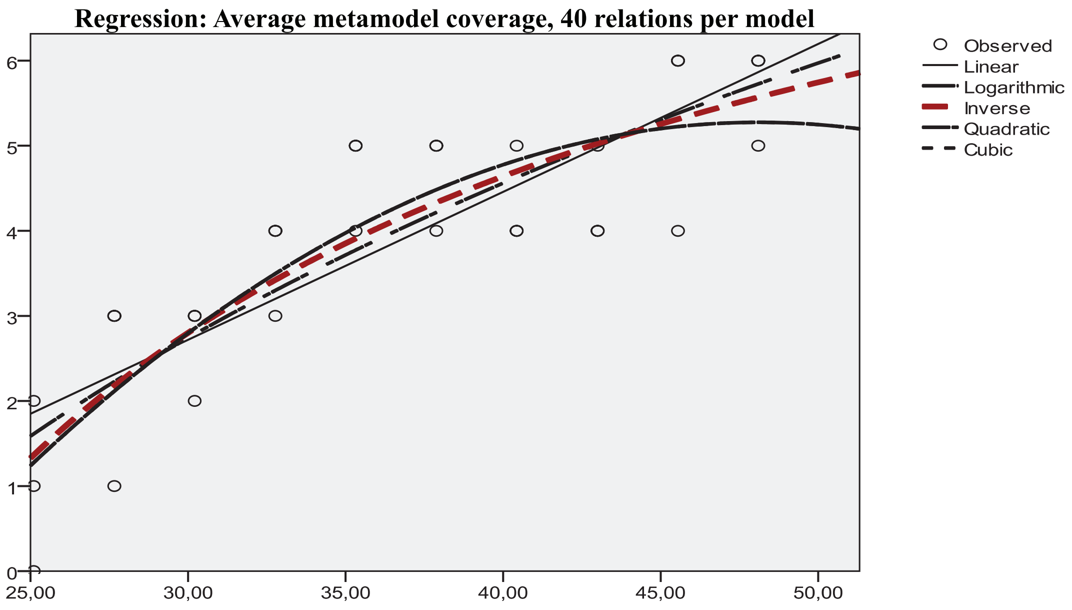


Figure 5.10: Visualisation of model estimates: Average metamodel coverage, 40 relations per model

Average Structure Coverage

Figure 5.19 shows the results of performing test runs with different average structure coverage and a relation count of 20 per model. In this figure, it looks like the errors increases almost linearly as the coverage increases. At the first two measurements points, no errors are found. Then the errors found increases up to measured coverage (7%). The highest number of errors found is 4.

Table 5.11 reveals that, while a cubic model fits best, a linear model fits the data almost just as well. The difference in the R^2 values between the linear curve and cubic one is only 0,5%. The quadratic model fits the data almost as well as the cubic model, even though the curve differs somewhat from the cubic curve, as can be seen in figure 5.20, in that is assumes that the increase in errors occurs more at lower coverage values. The cubic models follows a more linear curve.

Figure 5.21 shows the results of performing test runs with different average structure coverage and a relation count of 40 per model. At the lowest coverage point, a mean value of 2 errors were found, while at the highest coverage the mean errors found is 6. Although it appears that the number of exposed errors increases as the coverage increases, this increase fluctuates between measurement points. Between 6,33% and 7,17&, the number of errors remains steady, while at 8% it decreases. At 10,5% the errors decreases again, compared to the errors found at 9,66% and this mean value stays the same up to 13%.

The data presented in 5.12 appears to show that there is no particular model, that has been tested using the regression analysis, that fits the data well. The best model to fit the data only explains 50,7% of the variance. Figure 5.22 demonstrates the models that have

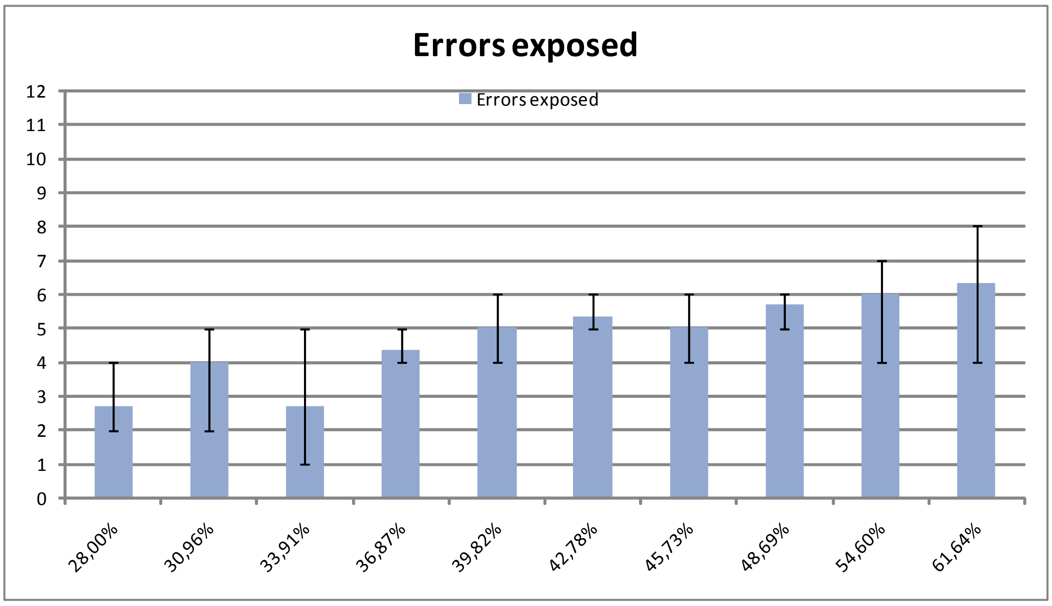


Figure 5.11: Errors exposed by sets with an average size of 60 relations with different average metamodel coverage.

	Model Fit	Parameter Estimates			
Equation	R^2	Constant	b1	b2	b3
Linear	,790	-,679	,570		
Logarithmic	,752	-,611	1,789		
Inverse	,595	2,922	-3,733		
Quadratic	,794	-,995	,774	-,026	
Cubic	,795	-,740	,496	,055	-,007

Table 5.11: Summary of model fit and parameter estimates, average structure coverage, 20 relations per model

been tested on the data.

In figure 5.23 the results are shown of performing test runs with different average structure coverage and a relation count of 60 per model. The results illustrate a fairly consistent rise in the number of errors, as the coverage increases. Still, at 11% to 12,38% coverage, the errors decreases, which also happens slightly at 15,51%, even though the error is relatively high for that coverage.

Table 5.13 illustrates that it is quite difficult to find a model that correctly describes a relation between coverage and the number of errors. The model that best fits the data, which is the cubic model, only explains about 42% of the variance, which is only marginally better than the linear model, which explains about 41% of the variance. Figure 5.13 shows that

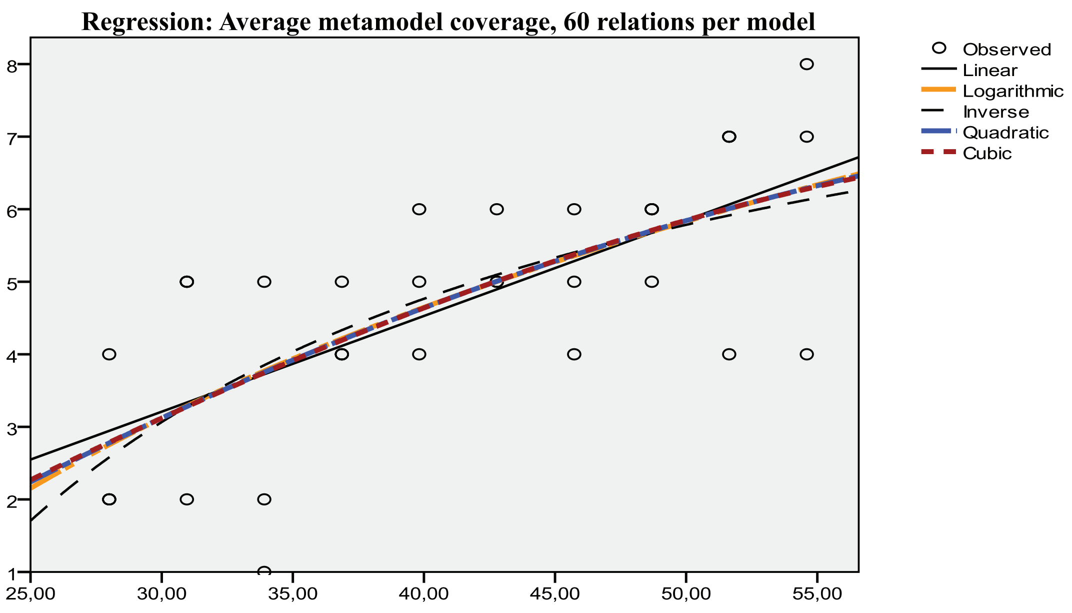


Figure 5.12: Visualisation of model estimates: Average metamodel coverage, 60 relations per model

	Model Fit	Parameter Estimates			
Equation	R^2	Constant	b1	b2	b3
Linear	,486	,290	,419		
Logarithmic	,503	-4,053	3,755		
Inverse	,507	7,804	-31,263		
Quadratic	,504	-2,764	1,127	-,038	
Cubic	,504	-2,764	1,127	-,038	,000

Table 5.12: Summary of model fit and parameter estimates, average structure coverage, 40 relations per model

all the models are quite similar to the linear model, and seem unable to explain most of the variance found in the data.

5.4 Performance

For the experiments, 28.000 models were generated using the model generator, and 22.000 models were tested on ACA.NET. Here we portray the time needed to perform the tests on the SF. The computer that has been used to perform all the experiments has a T7500 Intel® Core™ 2 Duo processor, which runs at 2.20 GHz clock speed. The computer has 4,00 GB of internal memory available. The platform is 32-bit Windows Vista™ Enterprise edition.

The first step of the testing process was generating 28000 models. Generating 10000

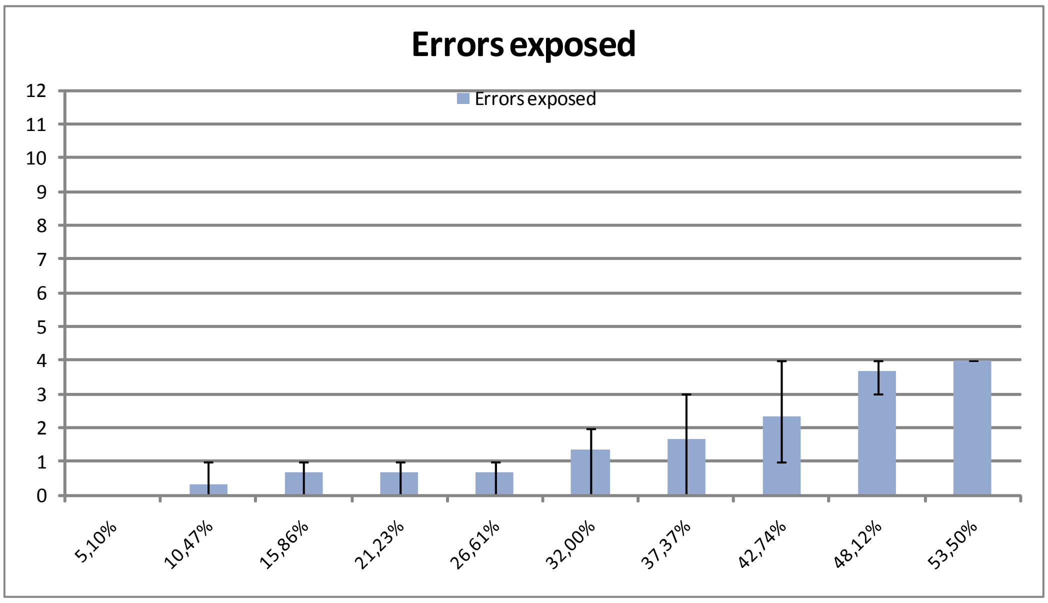


Figure 5.13: Errors exposed by sets with an average size of 20 relations, with different total structure coverage.

	Model Fit	Parameter Estimates			
Equation	R^2	Constant	b1	b2	b3
Linear	,414	1,093	,288		
Logarithmic	,401	-4,076	3,514		
Inverse	,381	8,147	-40,604		
Quadratic	,423	3,557	-,116	,016	
Cubic	,424	2,935	,059	,000	,000

Table 5.13: Summary of model fit and parameter estimates, average structure coverage, 60 relations per model

models took about 1 to 2 seconds, depending on model size. Validating each model (and the ACAFactory CVC rejecting some models, because of incomplete custom hooks), takes about 5 to 10 seconds, depending on the model size (the larger the model, the longer the validation takes to complete). Therefore, generating and validating models so that we ended up with a set of 6000 valid models took about 16 to 24 hours. However, if there was more time to create custom hooks that generated more valid models (about half the models were regarded as being invalid), then the generation process could have taken half the time.

The second step involved running all test sets through the ACAFactory. Testing a set of 100 models typically took up 25 to 30 minutes, depending on the model size. Running a single model through the ACAFactory and generating code from the model takes about

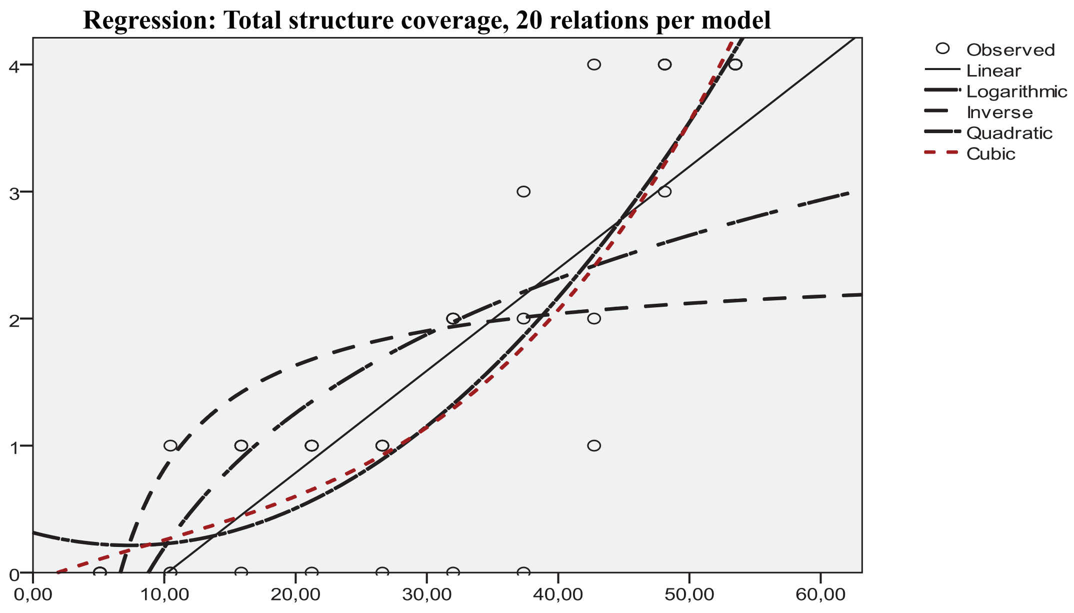


Figure 5.14: Visualisation of model estimates: Total structure coverage, 20 relations per model

10 to 30 seconds complete. test producing an error does not take up more time than a test that exposes no errors. Validating each model and generating code from the model took up most of the time. In addition, much time was needed for shutting down VS processes. These processes needed to be shut down because of several errors that could occur in the VS program. First, even though the process should remain entirely in the background, dialogs (popups) regarding ACA.NET still come up, often halting the testing process. A special watchdog timer was needed to ascertain whether VS was available for continuing the testing process. If not, VS had to be shut down and a new process started to take its place. Second, VS often crashed, resulting in an error-dialog explaining that VS had shut down. This dialog not only halts the testing process, it also renders the watchdog timer useless. Therefore, the system program that makes the error-dialogs appear was removed from the computer so that it could not halt the testing process. Testing can be much faster if these problems do not occur.

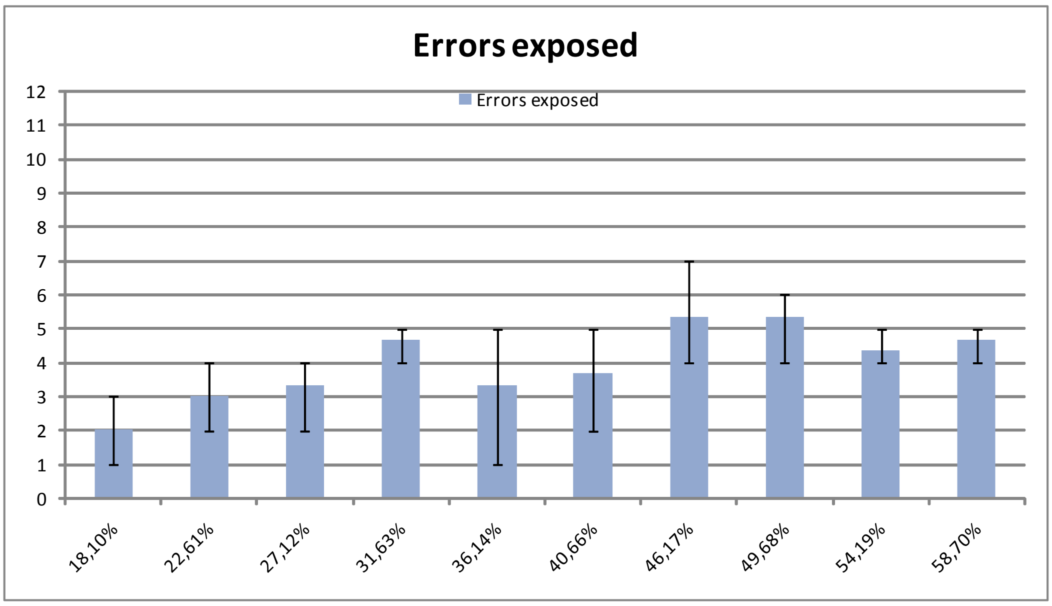


Figure 5.15: Errors exposed by sets with an average size of 40 relations, with different total structure coverage.

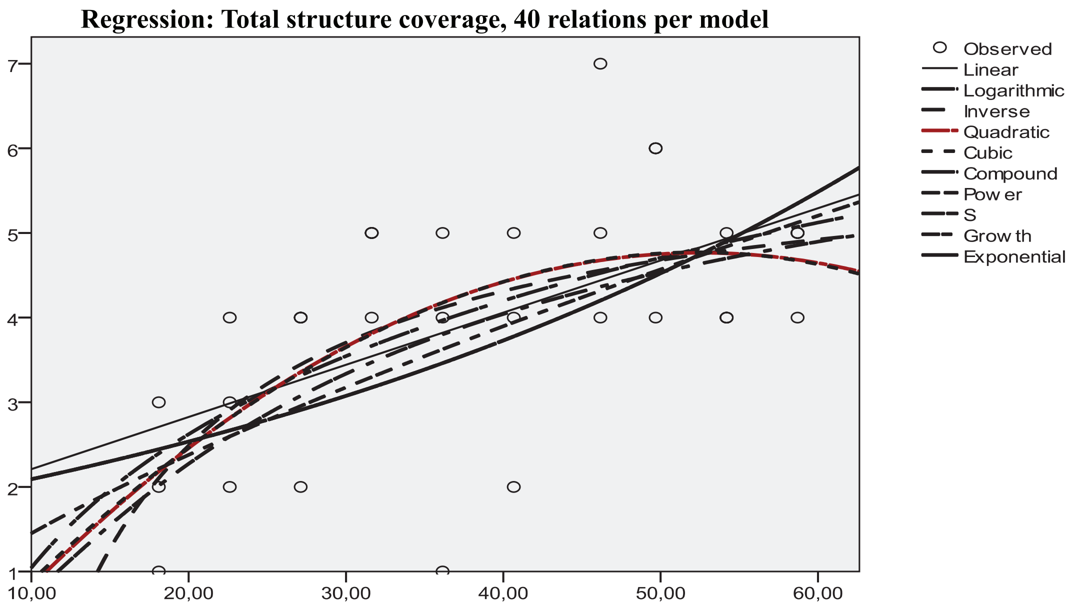


Figure 5.16: Visualisation of model estimates: Total structure coverage, 40 relations per model

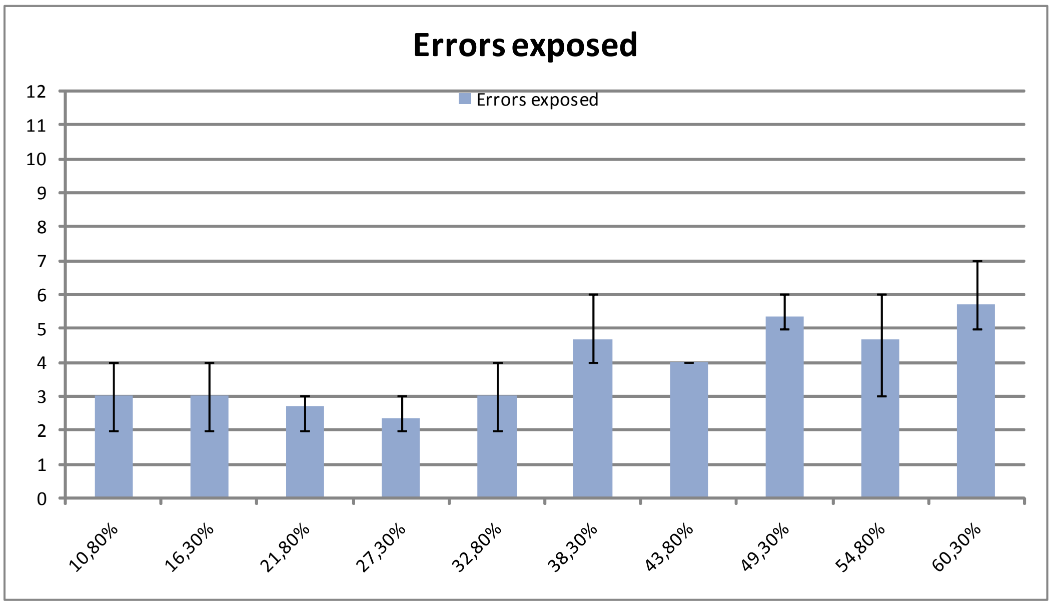


Figure 5.17: Errors exposed by sets with an average size of 60 relations, with different total structure coverage.

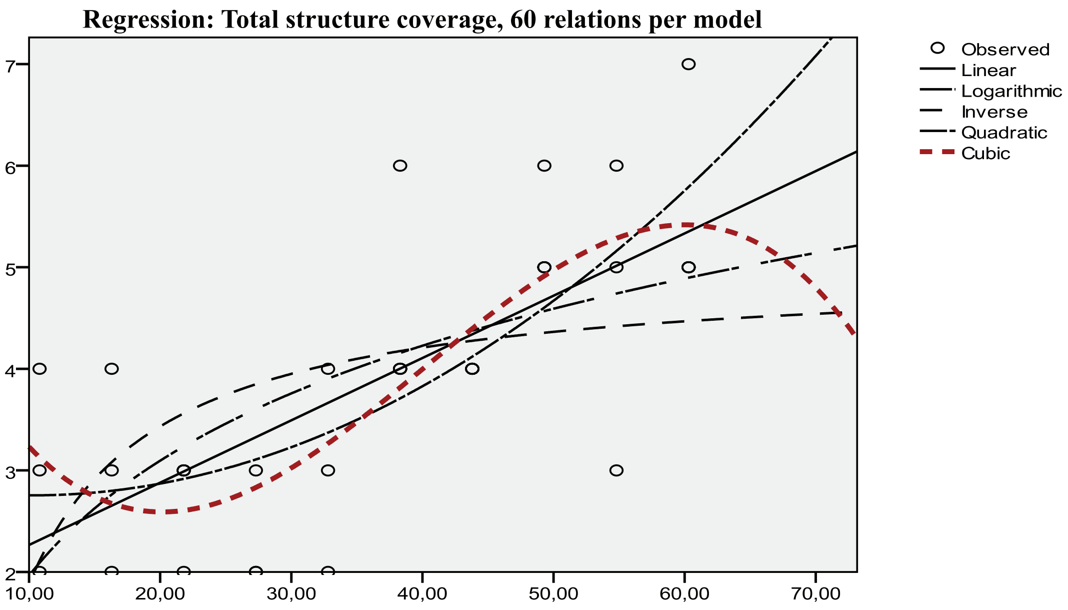


Figure 5.18: Visualisation of model estimates: Total structure coverage, 60 relations per model

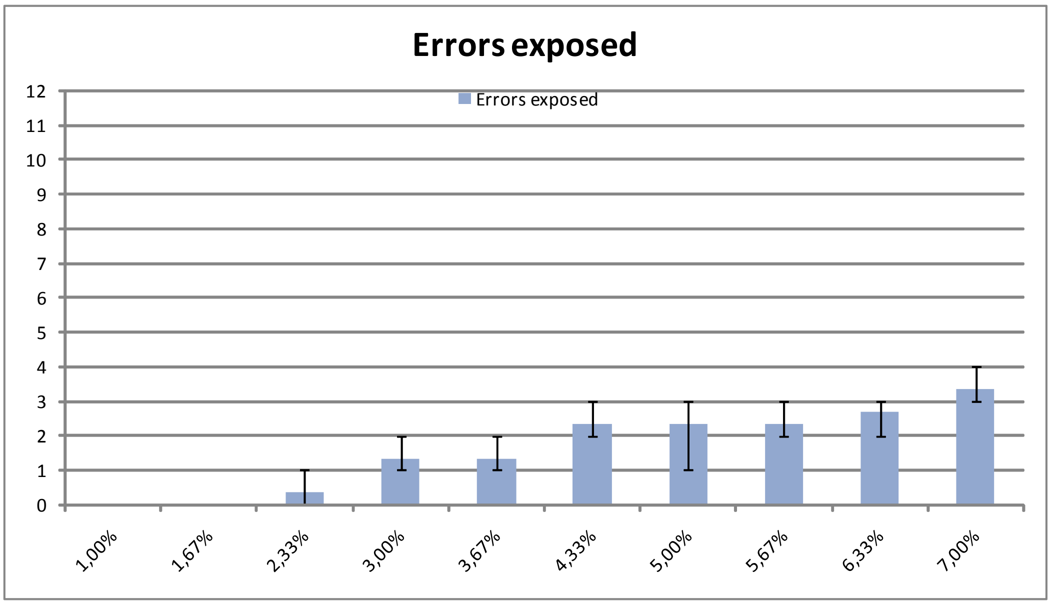


Figure 5.19: Errors exposed by sets with an average size of 20 relations, with different average structure coverage.

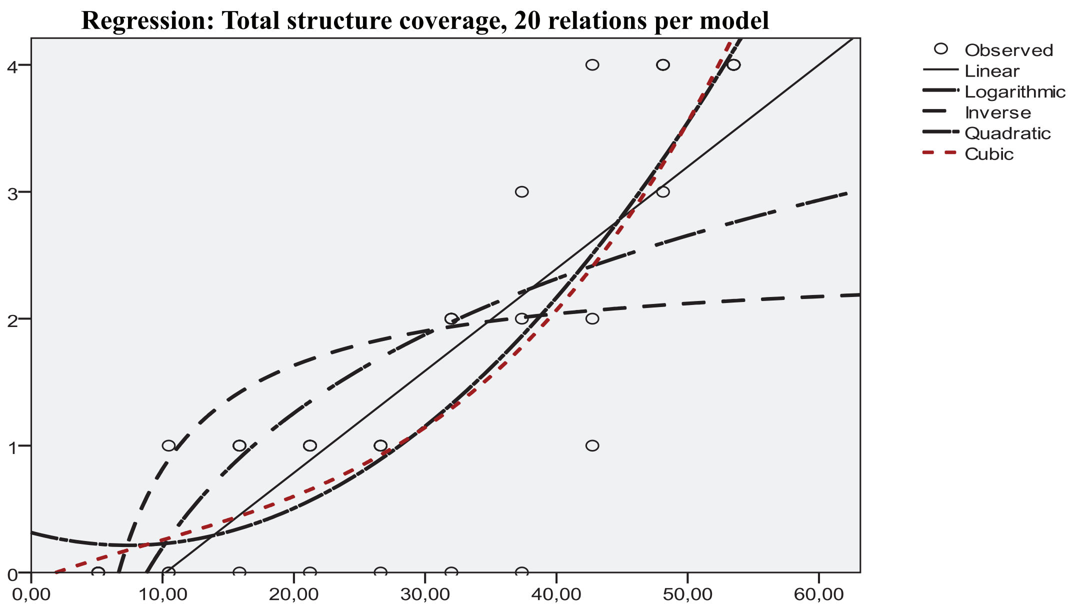


Figure 5.20: Visualisation of model estimates: Average structure coverage, 20 relations per model

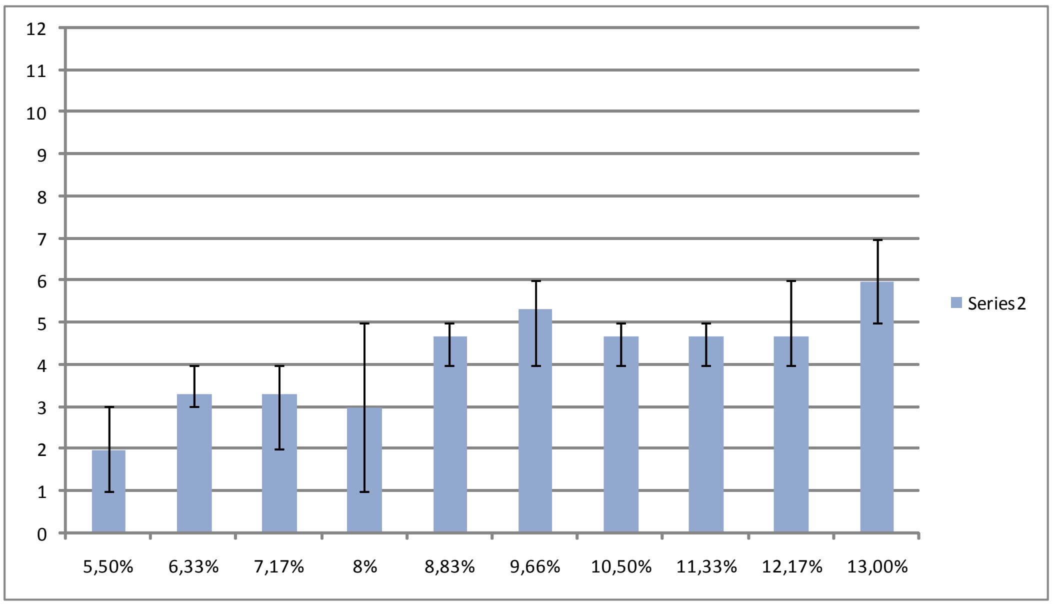


Figure 5.21: Errors exposed by sets with an average size of 40 relations, with different average structure coverage.

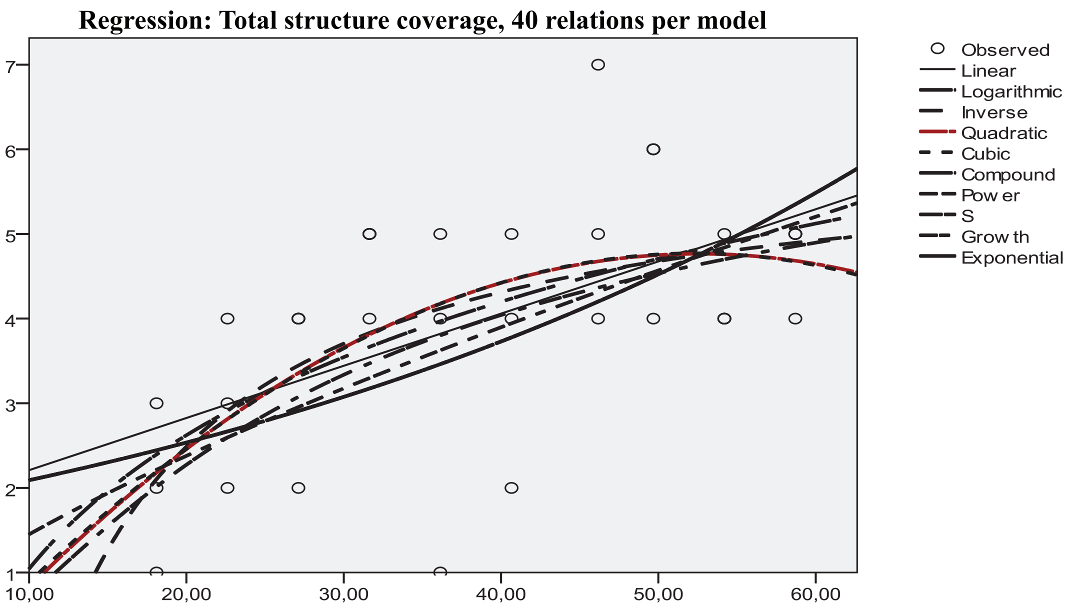


Figure 5.22: Visualisation of model estimates: Average structure coverage, 40 relations per model

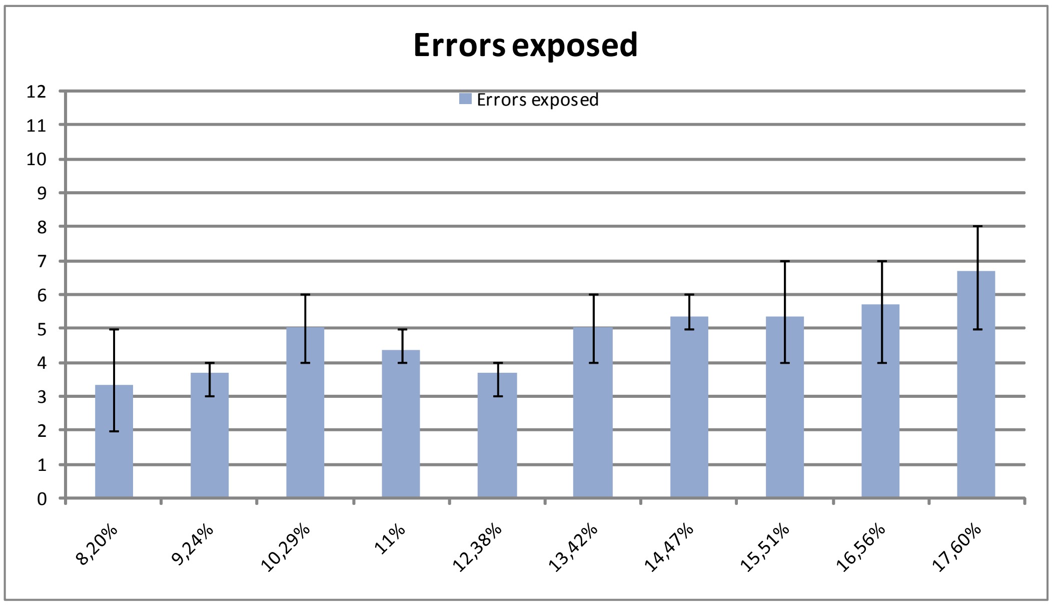


Figure 5.23: Errors exposed by sets with an average size of 60 relations, with different average structure coverage.

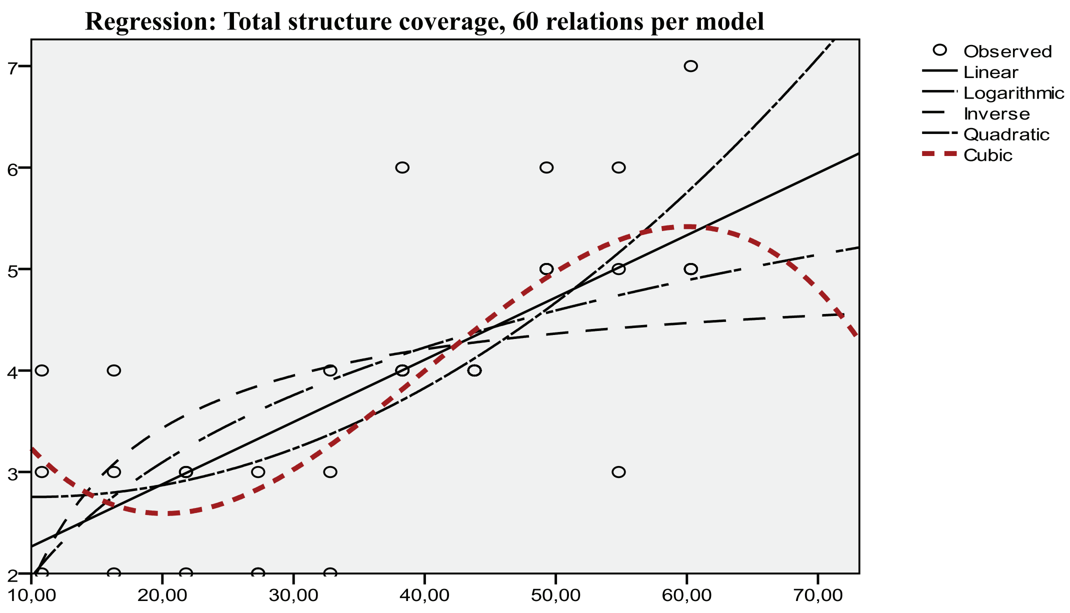


Figure 5.24: Visualisation of model estimates: Average structure coverage, 60 relations per model

Chapter 6

Discussion

In this chapter, the findings of the previous chapter are discussed. First the signifance of the exposed errors in the ACAFactory are discussed. After that, we discuss whether a proposed coverage method is fit to quantify test set quality. Finally, we consider the performance of the testing process.

6.1 Exposed Errors in the ACAFactory 5.1

The first part of the experiments aimed to find as many errors in the ACAFactory as possible and to determine what type of errors can be found using automatically generated tests. A total of 12 errors were found by testing the ACAFactory with 22000 models. Eight of those errors occurred during the generation process, while 4 occurred during the compilation of the generated code. Eight of these errors are caused by models being marked as valid by the Custom Validation Code (CVC), which should have been rejected instead. Only two errors can be reproduced by a developer that uses the visual designer (editor) that comes with the ACAFactory.

In our opinion, even though many of these errors cannot be reproduced, attempting to transform an invalid model into code should not crash the ACAFactory code generator. Apparently, the code generator crashes because the CVC does not verify that all constraints of a model are satisfied. This implies that the CVC relies on the existence of the editor, and that the editor correctly imposes limits on the internal structure of models. Even if the editor does correctly prohibit developers from creating invalid models, there are risks to such a system design.

Model validation is a single concern, yet it is divided over multiple parts of the ACAFactory. In addition, constraints that are handled by the editor are not written in code as being constraints, but rather as part of the user interface functionality. Then, just like CVC, these (hidden) constraints are expressed in C#-code, instead of a declarative language such as OCL [7] that promptly describe rules for models. This makes it harder for humans to understand what constraints are written in the editor-code. Also, with the evolution of the ACA.NET software, the editor is susceptible to change. With that change, it becomes possible that the editor no longer correctly imposes limits on model structures, so that currently hidden validation shortcomings (validation that is not part of the CVC, but part of the editor) are exposed to the developer.

According to [4], a robust system is able to handle situations that fall outside the system specifications. In this case, the use of a generator to create models is a situation not considered by the specifications of the ACAFactory and the ACAFactory does not respond well to it. Hence, the errors that are the result of ineffective CVC can be considered to be *robustness errors*. In that sense, the approach for automated test generation has primarily revealed robustness errors in the ACAFactory 5.1.

It is important to note that the ACAFactory itself has been thoroughly tested using traditional testing approaches, and used already in multiple projects. This indicates that the errors that were found using automated test generation are errors that are difficult to find by using traditional software testing approaches. However, it remains unknown whether other errors, which are easier to find using traditional testing approaches, can be exposed using the method of automatic test generation.

6.1.1 Exposed Errors in the ACAFactory 6.0

After exposing and identifying errors in the ACAFactory 5.1, test runs were performed on a later version of the ACAFactory. From the results of those test runs, 5 errors could be identified as being present in both the 5.1 as the 6.0 versions of the ACAFactory. These findings suggest that the errors found using the model generator are indeed difficult to find using traditional testing approaches. Yet, these findings also seem to suggest that over half of the errors found in ACA.NET 5.1 are not present in the later version. Conversely, it could be that with more tests, more errors could be exposed in the ACAFactory 6.0 that were also present in the ACAFactory 5.1. In other words, it is difficult to ascertain whether there are even more errors that are present in both ACA.NET 5.1 and ACA.NET 6.0. Also, the metamodel of the ACA.NET 6.0 tool is different than that of ACA.NET 5.1, and thus has different generator-code. Therefore, it could be that some errors were removed only because of changes in the generator-code, and not because the errors were found when testing ACA.NET 6.0.

6.2 Relation Coverage and Exposed Errors

The focus of the second part of the experiments is to find out whether any of the proposed coverage methods provide a good way for quantifying test quality. In this section, the findings related to this question are discussed. First, experiments were performed to find out if there is a relation between metamodel coverage and the number of exposed errors in the ACAFactory 5.1. Then, experiments were performed to find out the relation between structure coverage and the number of exposed errors.

6.2.1 Metamodel Coverage vs Exposed Errors

For models with a relation count of 20 or 40 relations, there seems to be at least a linear relation between metamodel coverage and the number of exposed errors. However, considering the data, one might consider a linear relation to be too primitive to properly describe the relationship. Most likely, the relation is of a higher order. According to the proportion of explained variance, a cubic relation could be a correct model for describing the relation between metamodel coverage and the number of exposed errors. Yet, other relations, like quadratic or sometimes even an inverse relation, fit the data almost just as well or better. Therefore, given the acquired data, it would be naive to conclude that the cubic model is the correct one. More statistical data and analysis is needed to determine whether such a model correctly describes the relation between metamodel coverage and exposed errors.

Even if we assume that a cubic relation is a model that correctly describes the relationship in question, then the magnitude of the relation is still important. The magnitude of a relation denotes how the dependent variable (errors) changes per change in the independent variable (coverage). If the magnitude is small, then a change in coverage only marginally increases the number of exposed errors. In the case of metamodel coverage (both total and average), the relationship does not appear to be of a small magnitude, as the average number of errors found increases up to 3 to 6 times as the coverage increases. Yet, the data points towards average metamodel coverage having a larger relation than total metamodel coverage.

The relation between metamodel coverage and errors, that was observed for model sizes of 20 to 40 relations, was not observed for models with an average relation count of 60. Although the average in errors rises as the average metamodel coverage increases, the relationship between average metamodel coverage and errors is vague. The relation between total metamodel coverage and the number of errors found is even less apparent, indicating that average metamodel coverage is also a better predictor for test set quality for large models.

Thus, it seems that aiming for high average metamodel values for test sets should improve the quality of the testing process, when testing the ACAFactory 5.1 using automated model generation. Yet, the relation between coverage and quality seems to be dependent on the size of the models that are used as test cases.

6.2.2 Structure Coverage vs Exposed Errors

Model size also seems to be an important factor for the relation between structure coverage and the number of exposed errors. When using test sets consisting of small models (i.e. with a relation count of 20 per model), there seems to be at least a linear relation between coverage and the number of exposed errors. Just like metamodel coverage, a cubic relation could be a possible model for describing the relation between structure coverage and the number of exposed errors. Yet, other relations seem to fit the data almost just as well. Therefore, it would again be naive to regard the cubic model to be the correct one and additional stastical data and analysis is required before determining what model correctly describes the relation between structure coverage and exposed errors. In test sets with small models, the data points towards total structure coverage having a somewhat weaker relation than average structure coverage.

In test sets with larger models (i.e. with a relation count of 40 or 60 per model), there seems to be no particular relation between structure coverage and the number of exposed errors. In other words, the number of exposed errors does not seem to rely on the amount of average or total structure coverage. In addition, the relation between structure coverage and the number of errors found appears to be quite smaller than that of metamodel coverage and the number of errors found.

6.2.3 Influence of Model Size

For all coverage criteria, the relation between coverage and errors appears to be weaker for large models. There are several possible reasons as to why the size of a model is important for the (pe43rceived) effectiveness of coverage methods. First, a single model can only expose a single error. So, even though a model has the potential to expose more than one error, the code generator stops generating code after the first one is encountered. In other words, errors can be hidden by other errors in a model. Consequently, the results for test sets with large models may possibly be influenced by the fact that a model can only produce a single error. If a model could produce more than one error, then it might be that more different type of errors are found by the tests. As large models have more model elements, and test sets with large models seem to expose more errors, we assume that such sets have a bigger chance of containing hidden errors than sets with smaller models. The consequence of hidden errors is that some luck is needed for a test set to expose many different types of errors. With no such luck, it could well be that only a few different errors are found, while the rest remains hidden. This would make it incorrectly appear that the test set is of low quality.

Second, as model size increases, so does the amount of coverage that can be achieved. Therefore, test sets with large models have higher coverage values. Perhaps the relation between coverage and errors is stronger for lower coverage values.

6.2.4 Comments on Regression Analysis

In the regression analysis of all the coverage methods, the results show that the highest proportion of explained variance is about 80%. There are several possible reasons that no higher value has been found in the results. First, it is possible that the error in the data is an artifact of the methods used to test the effectiveness of coverage methods. Yet, another reason for high percentages of unexplained variance is that the total number of errors found is only 12, and that most test sets only revealed about 4 to 6 errors. Therefore, a slight difference in the number of errors found can quickly lead to large error. So, having more errors in ACA.NET, or having test sets that are better able to reveal more errors in ACA.NET, could actually improve how well models fit the data.

Yet, these remarks also reveal that there are limitations of using only the proportion of explained variance to test whether a relation exists between coverage and errors. The method only gives an indication that there is a possible model that can fit the data. Additional statistical analysis will be required to be able to determine whether the model that seems to fit best is actually a valid model for the acquired data. Still, regression analysis can at least indicate whether a relation between errors and coverage actually exists and how strong that relation could be.

6.3 Performance of Model Generation and Testing

The aim of the project is to build a tool that helps in the testing process of software systems. This implies that the tool must be able to effectively find and report errors in a system under test. However, if the tool is too difficult to use, or it just takes up too much time to hunt for errors, then the tool probably will not be used, since using it will be less effective than using a traditional testing method. In section 5.4, it is noted that generating models takes very little time and that testing a model takes about 10 to 30 seconds, depending on the size of the model. Still, most of the time that it takes to use the generated approach to testing is because of the SF itself. If the SF has a better performance in validating a model, or generating code from a model, then it would improve the performance of the testing process to a great extent.

There is also the problem of the need for custom hooks. Custom hooks need to be programmed for each SF under test. Indeed, these custom hooks must be adapted for any change that occurs in the CVC. Building the custom hooks for the ACAFactory such that it would accept the majority of the generated models took up quite some time when compared to building the rest of the test software (generator and Test Chain (TC)). That is because we had to find out what constraints were imposed upon model by inspecting the factory code and by trial and error. Yet, a developer of a SF usually is aware of any constraints that are imposed upon models. This should make it easier, and less time-consuming, for the developer to implement the custom hooks.

6.4 Existing Approaches to Automated Test Generation for MDD

Even though the validation of model transformations is not a new concept, there is few recent work on the generation of test data for model transformators [30]. Still, some research has already been done on validating model transformations using automated test generation. Here we discuss our findings in the light of related research.

Brottier et al. [24] present an algorithm for generating UML models from the part of UML metamodel definition that concerns class diagrams. They propose several strategies for automatically generating test models. These strategies concern choices that need to be made at certain points during the creation of a model. An example of such a strategy is that objects should always be reused whenever possible when new relations are created. In our research, we were faced with the same choices that are listed by Brottier et al. The authors state that it is still unknown what strategies are better for creating high quality test models. Instead of using a predefined strategy, our approach was to randomly choose a strategy (e.g. randomly choose to create a new object or reuse another one). This way, we aim to generate as many kinds of test models as possible, and subsequently improve the testing method.

Kalinova et al. [41] use the idea of coverage based analysis and specification-based testing for generating test cases, generating test oracles and providing test coverage criteria for compilers. For generating test cases, the syntax production rules and constraints of a language are used to generate both a set of statically correct and incorrect programs.

Correct programs are denoted as positive test cases, while incorrect programs are denoted as negative test cases. In our research, this would correspond to positive test cases being valid models, while negative test cases would be invalid models. In our case study, using invalid models as test cases revealed robustness errors in the ACAFactory. Therefore, when testing SFs with an automatic model generator, intentionally adding invalid models to the test set improves its quality.

Fleurey et al. propose [29] a method for optimising test sets. They combine two methods for generating test sets, consisting of automatically generated models. First, they attempt to create a model that covers all possible metamodel elements. If the size of the model (in metamodel elements) grows beyond a predefined limit, another model is created to cover the remaining uncovered metamodel elements. This continues until all metamodel elements are covered by as few models as possible. Then, a bacterial approach [20] is used to optimise test sets, such that only models remain in the test set that contribute to the number of metamodel elements that are covered. This approach for qualifying test sets can be regarded as a total metamodel coverage approach. The results from our research indicate that the average metamodel coverage criterium is a better predictor for test set quality than total metamodel coverage. Therefore, the method that is used in [29] to optimise test sets may actually decrease the quality of sets.

In addition, the method used for testing a model transformator is an important point to consider when optimising or qualifying test sets. In our case study, the program under test halts immediately when it encounters an error during code generation. Models only produce one error, even if they have the potential to expose more errors. Thus, aiming to generate test sets with very few models (but with high metamodel coverage), does not necessarily improve the probability of exposing many errors in the model transformator.

Chapter 7

Conclusions and Recommendations

This chapter gives an overview of the project's contributions. After this overview, we will reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

7.1 Contributions

A significant issue with the traditional approach to designing and producing software systems concerns the cost and time to market of software systems. Software gets more and more complex, making design and realisation increasingly costly and difficult. This, in addition to the need for software developers to deliver products in increasingly shorter amounts of time, calls for new approaches for designing and producing software. An approach that confronts the problems of traditional software development is Model-Driven Software Development (MDD), which seeks to improve developers' productivity by imposing structure and standards to the modeling process and raising the level of abstraction at which software is developed. Also, it aims to automate the process of transforming high-level models into lower-level models or source code.

One of the tools that has been created to support the vision of software factories is Microsoft DSL Tools. It allows developers to define a modeling language, generate visual designers that are customized to a particular problem domain and generate code from models described in custom modeling languages. A combination of a modeling language, visual designer and code generator is called a software factory (SF). An example of a SF created using Microsoft DSL Tools is the ACA.NET 5.1 Factory, a framework that helps developers building service-oriented .NET applications by providing services, a set of reusable architecture components that accelerate the development process. In addition, it comes with a SF, built with DSL Tools, that developers can use to model and build their own service oriented applications.

In the field of software engineering, much effort is put into building reliable software systems. If such systems are not able to meet up with their requirements, the system becomes unreliable, produces incorrect results or could even become unsafe. To raise the reliability and quality of a program, software is usually tested using a wide range of testcases. Many approaches exist for testing software systems written in GPLs and verifying models and model transformations. Using these conventional approaches for testing software, it is possible to test whether generated source code from a specific model is reliable and conforms to the desired specifications of the developer. Yet, source code generated from a slightly different model can already be erroneous because of an error in the generation process. Because tests are generally created for a specific instance of a system, new tests have to be created for testing this different model, which might take considerable time and effort.

Our contribution is a tool that provides support for testing the transformation from model to code by a SF, using automated model generation. We implemented and described a tool that automatically builds test cases for DSL Tools solutions, optimizes generated test sets based on predicted test quality, automatically supplies the test set to a SF and gives an analysis of the results of the testing process. We validated the tool, and analysed its effectiveness, by using it to test the code generator of a SF that has been used in several 'real-life' projects. In addition, we tested the performance of four criteria for predicting test set quality.

7.2 Conclusions

The purpose of the thesis was to improve the testability of DSL Tools by extending it with a tool. The tool that has been built is a model generator that automatically generates input models for a SF that has been created using DSL Tools. First, the question was whether such a tools could be built. Then, we wanted to know how well the tool improves the testability of DSL Tools.

The result of the thesis project is a model generator that is able to generate test input for SFs, based on the SF's DSL definition. This model generator has been used to generate test input and several errors were found within ACA.NET. Thus, creating an automated test generation tool for SFs appears to be possible.

Looking back at the problem statements, presented in section 1.4, there are several questions that can now be answered regarding the additional value of using the test generation tool:

- *Do we find errors in ACA.NET that were not exposed earlier?* There are 12 errors found in ACA.NET 5.1 that were not exposed earlier. Eight of these errors are robustness errors, not reproducable by using the ACAFactorty editor, and therefore probably difficult to find using traditional software testing approaches.
- *Do we find errors in ACA.NET that are still present in a later version of ACA.NET?* The results of testing ACA.NET 6.0 was that there are at least 5 errors present in ACA.NET 6.0 that are also present in the previous major version of the tool, indicating that some errors found using the method of automated test generation are difficult to find using traditional testing approaches.
- *What kind of errors are found in the code generator?* An error has two subtypes, one specifies at what moment the error occurred, while the other indicates at what

step the cause lies for the error. For the first subtype, most errors (75%) occurred at the generation step. So, most of the errors found were generation errors, while the rest were compilation errors. The origin of nearly all errors was the validation step, implying that these errors occurred because of the validation code being unable to reject all invalid models. Therefore, most of the errors found were robustness errors.

Our conclusion is that an automated software testing approach is able to improve the testability of DSL Tools solutions. The approach is able to generate tests that appear difficult to reproduce by traditional testing methods and these tests do actually expose errors in a SF. The model generator is ready to be used to test any SF that has been created using DSL Tools. Yet, the possibility of CVC limits the ease with which a SF can be tested, as custom hooks need to be implemented to account for the additional constraints that CVC imposes on models. This would be less an issue if these constraints could somehow be part of the DSL definition, instead of being implemented as generator code.

Finally, we wanted to be able to quantify the quality of generated tests, so as to improve the quality of the tests and that of the software under test. Two coverage methods were proposed that could denote test set quality. The first coverage method that was tested is metamodel coverage. The definition of metamodel coverage is a combination of research done by Fleury et al. [30] and Wang et al. [56]. The second coverage method to be tested was structure coverage, which attempts to qualify models on the amount of possible permutations of model elements that are instanced. The test generator is able to select an optimized test set from a list of generated tests so that the test set attains high coverage values.

Test sets with several coverage values were created and supplied to ACA.NET and the number of different exposed errors were measured. These measurements have been analysed using nonlinear regression analysis so as to determine whether any relation exists between coverage and the number of errors found in ACA.NET. Also, the analysis can indicate how strong such a relation is. In the case of testing ACA.NET with an automated model generator, metamodel coverage proves to be a better quantity for qualifying test sets than structure coverage, as the relation between metamodel coverage and the number of errors exposed has a stronger magnitude in contrast to that of structure coverage. Also, the relation between metamodel coverage and errors seems to be less affected by size of the models, except when model sizes increases to about 60 relations per model. Average metamodel coverage performs better than total metamodel coverage. For any coverage method, the relation between coverage and errors appears to be smaller for sets with large models. This could be the result of errors being hidden by other errors in a model.From the results of the experiments, average metamodel coverage performed best in quantifying test set quality. Therefore, we recommend average metamodel coverage as the method for quantifying test set quality.

7.3 Future work

For the case study, a SF was chosen that had already been thoroughly used and tested. One of the reasons for choosing such a SF is that any errors that are found are most probably errors that are difficult to find using traditional testing approaches. Yet, there is no data that confirms that the method of automated test generation actually reduces the time needed to develop a DSL. Possible future work could aim to shed light on that subject. For example, a SF that is still in development could be an interesting case study. Both traditional, as well as automated testing methods, could both be used to test the SF and the time needed for both approaches could then be evaluated against each other.

Future work could also consider the coverage criteria and the prediction of test quality. First, one could obtain more statistical data regarding coverage and exposed errors by testing ACA.NET with more test sets. Also, it should be interesting to see whether the results, regarding the effectiveness of coverage criteria, are the same when testing SFs other than the ACA.NET Factory. Also, there are unanswered questions regarding the relation between model size and the performance of coverage methods for predicting test quality. In addition, several other methods exist for finding out how effective tests are. These methods include mutation testing [26] and other approaches that introduce errors in a system under test. These added errors can be used to find out if a testing method is able to reveal all the (mutation) errors in a system. Also, one can determine what kind of errors can be exposed in a system. Therefore, future research could focus on using such methods to determine how well automated testing is able to find errors in a system.

Other possible future work includes confronting the problem of metamodels not including all constraints in their definitions. One of the consequences of this problem results in the need for custom hooks to be able to generate valid models. Also, if it were possible to define constraints in a metamodel, then generating higher quality test cases becomes possible as it could be easier to assign meaningful values to properties. A language would be needed that has the power to incorporate many types of constraints. Furthermore, such a language could be incorparated into DSL Tools, which can then use the specified constraints for generating SF-code.

Bibliography

[1] A document that gives an overview of the visual t4 software, that aids in designing and creating text templates in visual studio 2005, or visual studio 2008, http://www.t4editor.net, 2009.

[2] Documentation about the resources that are part of visual studio extensibility (vmsx). one part of the vsx is visual studio sdk that allows the visual studio platform to be extended, http://msdn.microsoft.com/en-us/library/bb126235.aspx, 2009.

[3] Dsl tools lab - v1.0, a tutorial that enables the beginners to learn how to use the dsl tools in one day, http://code.msdn.microsoft.com/dsltoolslab, 2009.

[4] Eiffel software: ¨building bug-free o-o software: An introduction to design by contractTM¨, http://docs.eiffel.com/book/platform-specifics/exception-mechanism, 2002.

[5] Introduction of avanade connected architectures, http://www.avanade.com/delivery/acanet, 2009.

[6] The msdn library, a library of information for developers using microsoft® tools, products, technologies and services, a collection of documents that explains how to generate artifacts by using text templates, generating artifacts by using text templates, http://msdn.microsoft.com/en-us/library/bb126445.aspx, 2009.

[7] Omg's specification of the object constraint language (ocl) http://www.omg.org/technology/documents/formal/ocl.htm, 2009.

[8] OmgTMhome page, omg is an international computer industry consortium, that develops standards for a wide range of technologies and industries, http://www.omg.org/, 2009.

[9] An online tutorial from w3schools about xpath, a language for finding information in an xml document, http://w3schools.com/xpath/default.asp, 2009.

[10] An online tutorial from w3schools about xsl, a language for transforming xml documents, http://w3schools.com/xsl/default.asp, 2009.

[11] Overview of the iso 9126 standard to software quality assurance, http://www.software-quality-assurance.org, 2009.

[12] Spss statistics: Predictive analytics software, http://www.spss.com/statistics/, 2009.

[13] Uml® resource page, a collection of documents about uml and its specification, http://www.uml.org/, 2009.

[14] Documentation that comes with the software of microsoft visual studio 2008 dsk, microsoft visual studio 2008 sdk documentation, 2008.

[15] Demir A. Comparison of model-driven architecture and software factories in the context of model-driven development. *MBD-MOMPES '06: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, Washington, DC, USA*, pages 77–83, 2006.

[16] P. Klint A. van Deursen and J Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, pages 35(6):26–36, 2000.

[17] ASM. Introduction into abstract state machines, gives a brief overview of the concept of asm and the advantages of using the concept for describing algorithms, http://www.eecs.umich.edu/gasm/intro.html, 2008.

[18] Thomas Atkinson C., Kuhne T. Model-driven development: A metamodeling foundation. *Software, IEEE.*, 20(5):36–41, 2003.

[19] Karsai G.-Sztipanovits J. Neema S. Balasubramanian K., Gokhale A. Developing applications using model-driven design environments. *Computer*, 39(2):33–40, 2006.

[20] Fleurey F. Jezequel J.-M. Le Traon Y. Baudry, B. Automatic test cases optimization using a bacteriological adaptation model: Application to .net components. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 253–256, Washington, DC, USA, 2002. IEEE Computer Society.

[21] F. Bazzichi and I. Spadafora. An automatic generator for compiler testing. *Software Engineering, IEEE Transactions on*, SE-8(4):343–353, July 1982.

[22] Kurtev I.-Valduriez P. Bezivin J., Jouault F. Model-based dsl frameworks. pages 602–616, New York, NY, USA, 2006. OOPSLA 2006.

[23] Saleh K. Boujarwah A. S. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997.

[24] Steel J.-Baudry B. Le Traon Y. Brottier E., Fleurey F. Metamodel-based test generation for model transformations: an algorithm and a tool. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 85–94, Washington, DC, USA, 2006. IEEE Computer Society.

[25] Jones G. Kent S.-Wills A.C. Cook, S. *Domain-specific development with visual studio dsl tools*. Addison-Wesley Professional, 2007.

[26] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[27] EMF. A document that gives an overview of the eclipse modeling framework (emf) project, which is a modeling framework and code generation facility for building tools and applications, http://www.eclipse.org/modeling/emf/, 2009.

[28] Motwani R. Feder T. Clique partitions, graph compression and speeding-up algorithms. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 123–133. ACM, 1991.

[29] Baudry B. Fleurey F., Steel J. Validation in model-driven engineering: testing model transformations. In *Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on*, pages 29–40, Nov. 2004.

[30] Muller P.-A. Traon Y.L. Fleurey F., Baudry B. Qualifying input test data for model transformations. *Software and Systems Modeling*, 8(2):185–203, Apr 2007.

[31] Dinh-Trong T. Solberg A. France R.B., Ghosh S. Model-driven development using uml 2.0: promises and pitfalls. *Computer*, 39(2):59–66, Feb. 2006.

[32] Raymond K. Steel J. Gerber A., Lawley M. and Wood A. Transformation: The missing link of mda. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*.

[33] Cook S. Kent S. Greenfield J., Short K. Software factories. *Lecture notes in computer science*, 2004.

[34] Short K. Greenfield J. Moving to software factories. *Software Development, http://www.sdmagazine.com*, 2004.

[35] Short K. Greenfield J. Software factories: Assembling applications with patterns, models, frameworks and tools. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, New York, NY, USA, 2004. ACM.

[36] Tarr P. Hailpern B. Model-driven development: the good, the bad, and the ugly. *IBM Syst. J.*, 45(3):451–461, 2006.

[37] Bezivin J. *In Search of a Basic Principle for Model Driven Engineering*. Springer Berlin / Heidelberg, 2006.

[38] McCabe T. J. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[39] Küster J.M. Systematic validation of model transformations. *WiSME'04(associated to UML'04)*, 2004.

[40] Berkenkotter K. Using uml 2.0 in real-time development: A critical review. *SVERTS Workshop at the UML 2003 Conference, http://www-verimag.imag.fr/EVENTS/2003/SVERTS/*, 2003.

[41] Petrenkoa A. Posypkina M. Shishkova V. Kalinova A., Kossatcheva A. Coverage-driven automated compiler test suite generation. *Electronic Notes in Theoretical Computer Science*, 82:500–514, 2003.

[42] Zubin H. Lyu M.R., Cai S.K.S. S., and Xia. An empirical study on testing and fault tolerance for software reliability engineering. In *In Proceedings 14th IEEE International Symposium on Software Reliability Engineering (ISSRE2003*, pages 119–130, Nov. 2003.

[43] Bieman J. Karcich R. Malaiya Y., Li M. Software reliability growth with test coverage. *IEEE Transactions on Reliability*, 51:420–426, 2002.

[44] Uhl A. Weise D. Mellor S.J., Scott K. Model-driven architecture. *Lecture Notes in Computer Science, Advances in Object-Oriented Information Systems*, pages 233–239, 2002.

[45] Heering J. Mernik M. and Sloane A. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[46] Leavens G.T. Modified from Cheon Y. A simple and practical approach to unit testing: The jml and junit way. in boris magnusson. *ECOOP 2002*, 2374 of LNCS:231–255, 2002.

[47] Caruso J. Piwowarski P., Mitsuru O. Coverage measurement experience during function test. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 287–301, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[48] P. Pudlák, V. Rödl, and P. Savický. Graph complexity. *Acta Informatica*, 25(5):515–535, 1988.

[49] Binder R. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, Boston, MA, USA, 1999.

[50] Dunn R. *Software quality: concepts and plans*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[51] Wagner S. *Cost-Optimisation of Analytical Software Quality Assurance, PhD Dissertation*. PhD thesis, Technische Universitt Mnchen, 2007.

[52] Soukup J. Soukup M. Conference on object oriented programming systems languages and applications. *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 753–756, 2007.

[53] Lawley M. Steel J. Model-based test driven development of the tefkat model-transformation engine. *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 151–160, Nov. 2004.

[54] Feige U. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[55] Brown A. W. Model driven architecture: Principles and practice. software system model. *Software and Systems Modeling*, 3(4):3:314–327, 2004.

[56] Kim S.-K. Carrington D. Wang, J. Verifying metamodel coverage of model transformations. In *ASWEC '06: Proceedings of the Australian Software Engineering Conference*, pages 270–282, Washington, DC, USA, April 2006. IEEE Computer Society.

[57] Kenneth I. Warren A. A complexity measure based on nesting level. *SIGPLAN Not.*, 16(3):63–74, 1981.

[58] S. V. Zelenov, S. A. Zelenova, A. S. Kossatchev, and A. K. Petrenko. Test generation for compilers and other formal text processors. *Program. Comput. Softw.*, 29(2):104–111, 2003.

[59] May J. Zhu H., Hall P. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

ACA.NET A framework developed by Avanade that includes a code generating tool

CA: Coverage Analyser

CVC: Custom Verification Code

DSL: Domain Specific Language

DSL Application: Application built using DSL Tools

DSL Tools: Allows developers to define a modeling language and generate code from models

GPL: General Purpose Language

MDD: Model Driven Development

MDA: Model Driven Architecture

OMF: Meta Object Facility

OMG: Object Management Group

SF: Software Factory

SOA: Service Oriented Architecture

TC: (Code Generator) Test Chain

UML: Unified Modeling Language

VS: Visual Studio