ModelScope

Jacob Bølling Hansen

Kongens Lyngby 2014
IMM-M.Sc-2014-

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc-2014-

Summary (English)

The goal of this thesis is to construct a tool to support the analysis of models by size metrics. This is achieved by devising and analysing application scenarios, implementing a tool to support them, and validating the scenarios with the tool.

The two main aspects of usage application for such a tool are identified as the exploration of potential metrics and the exploration of models using these metrics.

Our approach is based on the metaphor of an oscilloscope: imagine an electrical engineer trying to understand a circuit by probing its parts, measuring different dimensions on various scales and units, juxtaposing and dynamically visualising different channels. In analogy to this, imagine a software engineer trying to understand a model applying our tool, “ModelScope”: we offer a variety of base and derived metrics, flexible options for units and scales, interactive visualisations with a choice of graph types in a graphical user interface designed to reduce the entry barrier and invite exploratory behaviour, making ModelScope suitable for class room usage by students. With regards to researchers, ModelScope provides a metrics exploration perspective, and its modular and extensible architecture allows to easily add and modify metrics as needed.

Summary (Danish)

Målet for denne afhandling er at konstruere et værktøj til at assistere analyse af software modeller med størrelsesmetrikker. Dette opnås ved at opstille og analysere en række anvendelsesscenarier, implementere et værktøj til at supportere dem og validere scenarierne med værktøjet.

De to vigtigste aspekter af anvendelsesscenarier for et sådant tool, er identificeret som værende udforskningen af potentielle metrikker og udforskning af modeller ved hjælp af disse metrikker.

Vores tilgang er baseret på metaforen om et oscilloskop: man kan forestille sig en elektro-ingeniør der forsøger at forstå et kredsløb ved at sondre dens dele, måle forskellige dimensioner på forskellige skalaer og enheder samt sammeligne og dynamisk visualisere alt dette via forskellige kanaler. Med dette som analogi kan vi forestille os en softwareingeniør der analyserer en software model med værktøjet "ModelScope": vi tilbyder en bred vifte af direkte og indirekte metrikker, fleksible muligheder for enheder og skalaer, interaktive visualiseringer med et udvalg af diagramtyper i en grafisk brugergrænseflade, designet til at invitere til udforskende brug, hvilket f.eks. gør ModelScope brugbar i undervisningssituationer for studerende. Med hensyn til forskere, giver ModelScope forskellige perspektiver til at udforske nye metrikker og den modulære arkitektur gør det muligt og nemt at tilføje og ændre metrikker og visualiseringer efter behov.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfilment of the requirements for acquiring the MSc degree of Science. This thesis was prepared by me between September 2013 and January 2014 under the supervision of Prof. Dr. Harald Störrle. This thesis is worth 30 ECTS credit points.

Lyngby, 24-January-2014

Jacob Bølling Hansen

Acknowledgements

I want to thank my supervisor Harald Störrle for his support, help and guidance both regarding the particular topic and the process of writing a thesis in general.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Fenton writes in “Software Metrics - a Rigorous & Practical Approach” [FP98] that *“Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules”*. What this means is that to make a measurement we need to identify the attributes of something hitherto abstract and find appropriate metrics of quantifications to make them comparable. Thus, measurements are one of the foundations of empirical science. Most important it enables us to make comparisons. It makes us take something abstract like the statement *“I am taller than you”* and concretise it to *“My height in centimetres is a higher number than your height in centimetres”*.

How do we for instance measure a carton of milk? We would first have to identify attributes of the milk carton to measure. Quantity as volume or as weight? Quality? Genesis and organic properties? Colour of carton? Colour of milk? Taste? Freshness as time? Viscosity? Nutrition properties such as amount of vitamins, fat percentage, energy etc. Some of these are directly measurable by existing agreed-upon metrics such as volume in litres and freshness in time, but other attributes such as taste or quality is harder to quantify directly.

Fenton distinguishes between “direct” measurements and “indirect” or “calculated” measurements. “Direct” measurements has easily quantifiable properties such as length or weight. “Indirect” measurements are somehow derived from direct measurements. The quality of the milk would be an example of an indirect measurement and would thus be a product of many other (measurable) attributes of the milk, such as freshness, organic parts, viscosity etc. Eventually it might be possible to come up with some sort of weighted quantifiable measure for quality of milk, but it would have to be based on a rigid definition which again might not be universally accepted.

In software engineering, attributes such as quality or maintainability may not be directly quantifiable but trying to define metrics for these attributes is advancing our understanding of them. Direct measurements regarding for instance size of software projects enables us to make necessary assessments regarding cost, time frame, maintainability and complexity among many other indirect measurements. Well known metrics for the size of software project are for example number of lines of code and function points.

The amount of interesting things we can assess about software increases as we find new ways to measure it.

Model-Driven Architecture is a software design approach with guidelines for structuring software specifications through models. According to [CL07] *A standardized method for determining sizing concepts for software models that allows the effective base lining and comparison of model concepts is a crucial need within the MODELS community.*

If we want to measure a software model we can look at several more or less direct measurements. Some examples could be file size, number of model elements, canvas space, number of pages when printed and others - some more easily quantifiable than others and some again more refined than others. These are some of the more or less direct measurements of software models. The indirect measurements of software models are not that easily defined. As stated earlier, indirect measurements are combinations of direct measurements and a way of finding good indirect measures could be to visualise the direct measures. Right now, such good agreed-upon measures do not exist [CL07], and that is why we want to construct a tool that can aid the research on the topic.

1.2 Problem Statement

Software models contains a range of attributes that may be used as metrics for productivity, quality etc. but little research exists on the topic. It is the thesis of this project that a tool could help explore these aspects of software model metrics and therefore sets out to analyse the needs and usage scenarios of such a tool before developing it and testing it accordingly.

1.3 Scope and limitation

The approach of this thesis is to go through the following steps:

Define application scenarios A number of application scenarios are considered and two are selected for a more thorough investigation. These are then textually described as scenarios and reviewed later in the Validation (section 6) with regards to the constructed tool. The application scenarios are to be found in section 2.1.

Elicitate goals, features and information models Based on the above mentioned scenarios, goals, features and information models are extracted to serve as requirements for the software development process of ModelScope. Goals, features and information models can be found in the Requirement Analysis of section 3.

Create and explore prototypes Different prototypes are constructed and tested out to decide on a graphical user interface as discussed in section 4.1.

Decide on technological basis for implementation Section 5.1 reviews choices for technology for the implementation of ModelScope.

Implement tool ModelScope is implemented using the requirements defined in the above mentioned sections. The implementation is described in section 5.

Validate tool through revision of the application scenarios and features Lastly, ModelScope is tested with regards to the identified qualities and features, especially the application scenarios of section 2.1. Unit Testing and performance tests are to be found in section 5.4 while the application scenarios are validated in section 6.

This thesis will only focus on the structural parts of software models. I.e. ModelScope in the current version will not look at the graphical representations of the models such as canvas space, colours, connectors, placement of diagram elements etc. It should however be a doable task to implement such probes and modifications to the parsers, but this will have to be in future work.

The thesis will not try to give any qualitative answers about good metrics, nor will the tool hold any automated data mining facilities.

Regarding probes and visualisations The scope of this thesis does not include to implementations of a vast number of different probes and visualisations but rather to provide an expandable architecture that allows models researchers to come up with new probes, readings and visualisations and implementing them into ModelScope's framework.

1.4 Related Work

The area of software model metrics is not greatly covered by literature and few tools exists that addresses the issue. Fenton describes a range of software metrics in [FP98] but measurements of models seems to be a neglected area. In regards to tool support for model metrics, it seems that only some of the various UML editors provides a small degree of metric capabilities. All these are of course limited to models constructed within the respective tools.

1.4.1 Manually investigating metrics through counting and spreadsheets

The most obvious way to investigate models and their metrics, is simply to manually count the model elements and put them into structure via. spreadsheet tools like Microsoft Excel. This, however, is obviously not practical for all but the smallest models.

1.4.2 MagicDraw Metrics and other UML tools

Some UML tools like MagicDraw has metric (counting) features that allows for basic counting of elements and exports of the results. The tools are vendor specific and very limited in functionality.

One of the goals of ModelScope is to be able to come up with new probes progressively. MagicDraw actually allows for extensions, so it is possible that new metric-implementations would be doable. Unfortunately this would still be limited to only MagicDraw models, and maintaining the tool would depend on MagicDraw and their continuous support of the functionality.

Chapter 2

Domain Analysis

2.1 Application Scenarios

To get an idea of uses of software model metrics a number of application scenarios will be described. Some of them has basis in real life cases while others are hypothesised on basis of brainstorming. Each scenario explains a case where model metrics are used to achieve some result. The scenarios will be revisited in section 6 where they will be analysed with the tool support of ModelScope in mind.

The application scenarios listed in table 2.1 are a result of brainstorms and the scenario of section 2.1.1.1 is inspired by [Sto10]. A few select application scenarios will be discussed in detail in section 2.1.1.

2.1.1 Detailed Scenarios

Following the list of application scenarios in table 2.1, this section explores some scenarios in greater detail. The scenarios will serve as inspiration for goal and feature extraction in section 3.2 and 3.3.

Usage	**Short description**
Compare to other known size metrics of software	Compare software model based metrics to other known software size metrics such as lines of code.
Comparing sizes of software models	Simply compare the scale of two or more models. “Scale” or “Size” may be subjective and different metrics should be tested. Reasons for exploring model sizes of software models could be to proof a point in related research. The researcher may want to argue that a project is of a certain magnitude, but lack the appropriate metrics to claim such a thing. In that case, she may already have an idea of what defines her models as “large” and simply wants to illustrate a point using model size metrics. She may also have some other projects in mind, and use model size metrics to compare different models to use their relative size as an argument.
“Edit distance”	How different is this model from last version I looked at? Domain: Business, Research, Teaching
Explore direct and indirect metrics	Analyse combinations of direct measurements to inspire new indirect measures of software models. Examples could be • Use Cases pr. actor • Number of elements pr. diagram • Average number of references pr. class
Fingerprinting	Is it possible to come up with a measurement based metric which would work as a “fingerprint” for a software model? Maybe something that could be visualised as a DNA fingerprint or perhaps even a single metric or “hash” uniquely identifying the model with regards to its structure. Could be used in conjunction with “Plagiarism Detection”.
Plagiarism Detection	It could be useful to utilize a fingerprinting metric for plagiarism detection in a teaching situation.

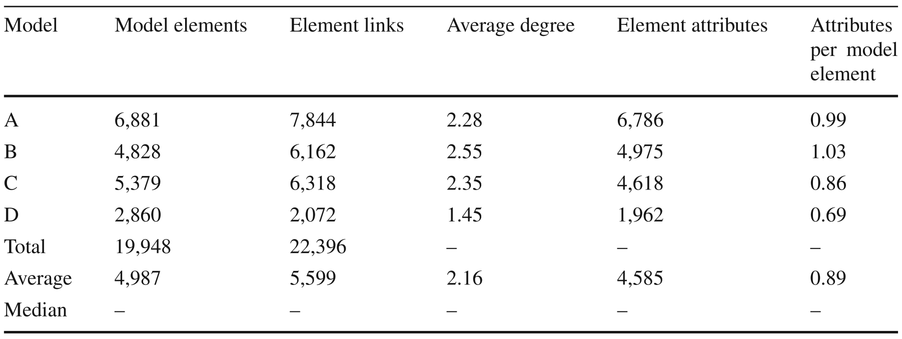
Table 2.1: Non-exhaustive list of possible application scenarios from brainstorm

Usage	**Short description**
Productivity analysis	To analyse the progress of several versions of the same model over time, could give information about productivity and efficiency as well as the evolution of complexity of the model involved.
Quality metrics	Attempt to find correlations between direct measurements, indirect measurements and known retrospective facts about software models. • Cost • Number of versions • Time to make (e.g. man hours) • Error rate
Software Model Research	Analyse structures in software models. Probe different models to examine where they resemble and where they differ.

Table 2.2: Non-exhaustive list of possible application scenarios from brainstorm (continued from table 2.1)

2.1.1.1 Model Research

This application scenario is based on the real life case that is the genesis of part of the paper "Towards clone detection in UML domain models" [Sto10]. In this paper Störrle investigates methods of defining similarities in models with heuristics for clone detection in mind. This is done by analysing the structure of four models. Data is collected by a program written specifically for that purpose from the models and then compared in tables (as in Figure 2.1) and visualisations (Figure 2.2).



Model	Model elements	Element links	Average degree	Element attributes	Attributes per model element
A	6,881	7,844	2.28	6,786	0.99
B	4,828	6,162	2.55	4,975	1.03
C	5,379	6,318	2.35	4,618	0.86
D	2,860	2,072	1.45	1,962	0.69
Total	19,948	22,396	–	–	–
Average	4,987	5,599	2.16	4,585	0.89
Median	–	–	–	–	–

Figure 2.1: Table from clone detection paper. Shows various metrics from the four models, such as number of elements, number of links and average number of attributes pr. element.

The measurements and comparisons allows the researcher to make observations regarding the four models structures. In this particular case, Störrle uses these observations to assess that the structure of models resembles that of a tree rather than that of a full graph. A lot of other observations about the models comprehensiveness in general can also be extracted from these visualisations of measurements.

Probing models like this is cumbersome work. It is time consuming and the degree of freedom with the metrics is therefore restricted.

The obvious case is the study of UML models, but it may be interesting to investigate other types of abstract models that fits the structure.

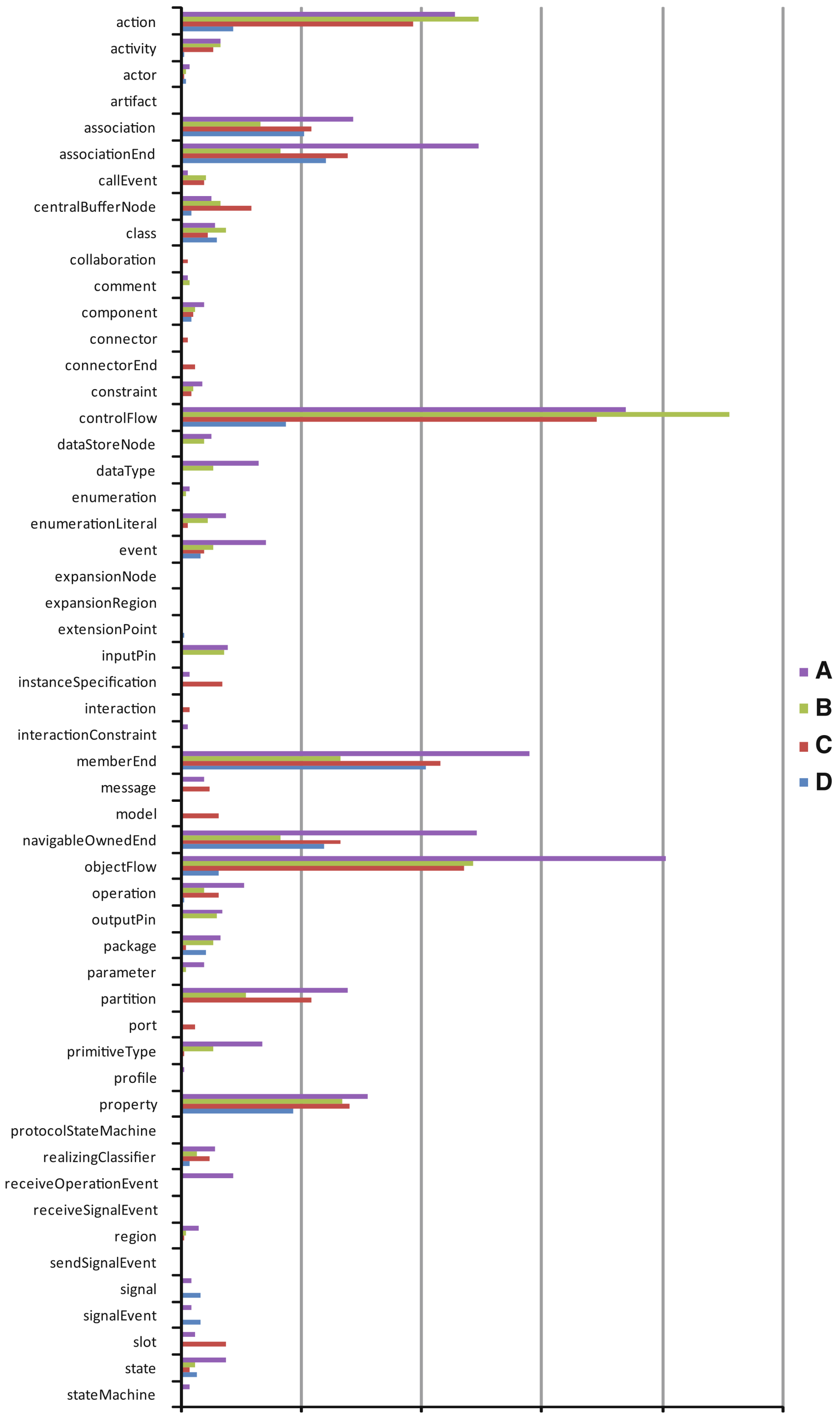


Figure 2.2: Visualisation of the metrics of the four models. See Figure 6.4 for a replication of this using ModelScope.

2.1.1.2 Metric Exploration

A model researcher could be working with the specific notion of model metrics and qualities of models. As we do not have consensus of what defines quality in a model, it could be interesting for a researcher to be able to compare different metrics of models and parts of models to investigate this aspect. These unknown factors requires a dynamic and versatile approach where the researcher would want to do all sorts of different kinds of probings on the models. She will want to explore and come up with different kinds of probes and readings and different ways of visualising the results. The work will be an iterative process of going back and forth between probing the models and visualising the results.

A typical process could look like this:

Firstly, the researchers sketches the model data and which parts to look at. The process begins with the model elements as the abstract cloud “M” in Figure 2.3 and works its way through splits in the data. These “splits” could be analogous to applying different probes at the model mass; it is what defines which part of the model to look at. At any level of this process, we can take a reading from the probe result (bottom level of Figure 2.3) and get an actual number, a count, or calculate a weighted result by some definition.

Secondly, the researcher will want to see the the results of the above visualised in some way. The visualisations would function as some sort of preliminary indirect measurements, or at least give indications of what those might be. Fenton describes “indirect measurements” as calculated from direct measurements, and in a way the visualisations does just this; they show correlations which is combinations of direct measurements.

Such visualisations may be sketched out first as in Figure 2.4 or done in some software. Likewise the different probings would have to be done by hand; an error prone and tedious task. Not only that, but also not a very flexible approach. As this process would be an iterative one it is likely (and hoped for) that each iteration will give rise to ideas for new probes or calculated measurements.

Tool support for this scenario would greatly improve the flexibility of probing and versatility of visualising.

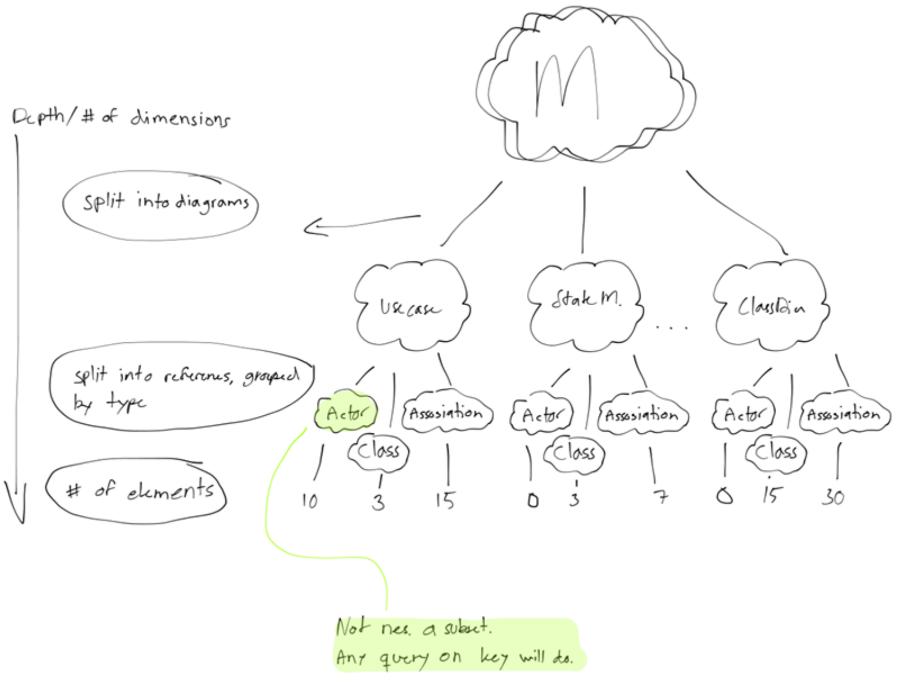


Figure 2.3: Sketch of the probe exploration process

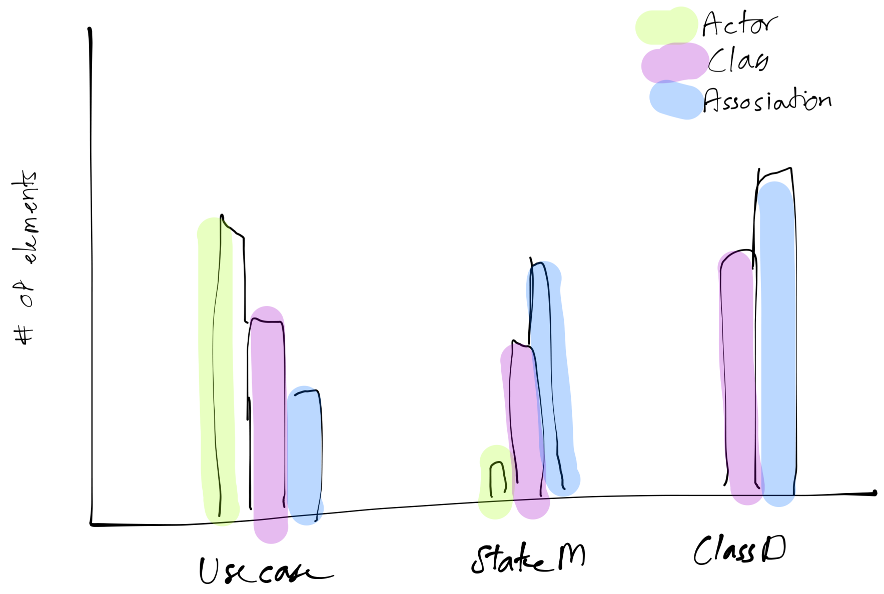


Figure 2.4: Sketch of a visualised result of probing

2.2 Terminology

This section explains some of the terms used in the thesis. Some of the terms are derived from the oscilloscope as a metaphor for a tool of visualising metric analysis while others are specific to ModelScope. The purpose of this section is to equip the reader with a basic understanding of what is meant by these terms. They will be explained in greater detail later in the thesis where it is required.

- **Model** A software model. Typically a UML model. The terms "model" and "software models" will be used interchangeably and considered synonymous unless specifically stated otherwise.
- **ModelScope** The name of the tool that is constructed in this thesis. Sometimes also referred to as just "the tool".
- **Visualisation** is a visual representation of `ProbeResults` and their readings. It could be a simple table, a bar chart or pie chart etc. A visualisation has "`Channels`" in which the results are visualised.
- **Channel** A `Channel` is a part of a visualisation wherein results of a `Probe` and it's corresponding `Readings` can be displayed. A `Visualisation` can have 1 or more channels. A channels depth is determined by the number of dimensions it can display. Each dimension will display a `ProbeResult`.
- **Probe** A `Probe` can be applied to a collection of model elements and returns a number of subsets ("`ProbeResult`") according to the rules prescribed in the `Probe`. An example of this could be to group all currently loaded model elements by element type i.e. "Use Case", "Actor", "Class", "Association" etc. Probes can be simple groupings or contain conditions about the output, such as "only diagrams". Note that the resulting subsets of probes can in turn be subjected to probes again. The number of times probes can be applied will be limited by the number of channels in the output-visualisation.
- **Probe Result** A Probe Result is a resulting subset of a probe being applied to a collection of model elements. Note that a Probe Result is itself a collection of model elements.
- **Reading** A Reading is a quantifiable measure that can be imposed on a subset of model elements (i.e. a "`ProbeResult`") created by a Probe. It could be "Number of elements" or something more advanced as "Average number of references pr. element".

Chapter 3

Requirements Analysis

3.1 Context

This section gives an overview of the neighbouring systems ModelScope will have to work with and the type of users likely to be interacting with it.

3.1.1 Neighbouring Systems

ModelScope will be a stand-alone application, but will have to operate on data from external applications. This section will give a brief introduction to some of the most important neighbouring systems in regards to ModelScope.

Figure 3.1 gives an overview of ModelScope and the neighbouring systems.

3.1.1.1 MagicDraw

MagicDraw is a visual editor for UML and other modelling languages. This is the primary tool ModelScope is expected to work with. ModelScope will focus

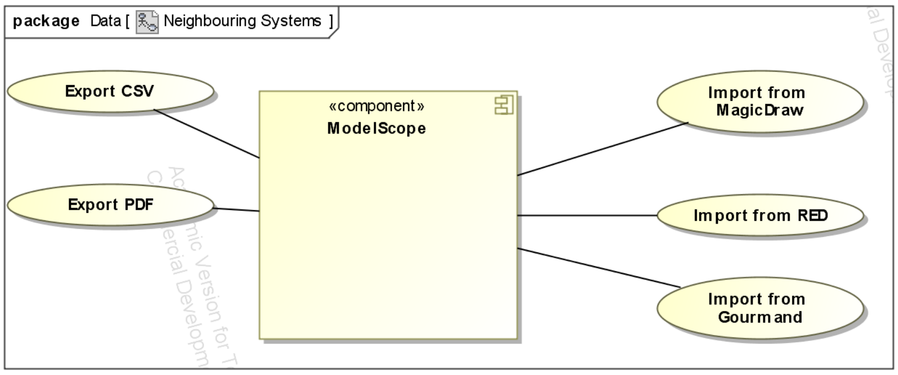


Figure 3.1: Use cases regarding the neighbouring systems of ModelScope.

primarily on the UML part of MagicDraw's capabilities. MagicDraw provides a variety of tools for UML support, such as code generation and reporting. MagicDraw has some abilities in regards to model metrics as well, but these comes down to counting and exporting counts. I.e. there is very little room for an exploratory approach to these metrics within MagicDraw. Hence an external tool such as Excel or other would be necessary to explore metrics exported from MagicDraw.

MagicDraw exports to MDXML file format, which is a XMI-document with proprietary additions. These additions are version specific.

3.1.1.2 RED

RED is a stand-alone requirements engineering tool. It is used in courses and a lot of development is going on as thesis projects and research work.

RED has certain modelling aspects that might be interesting to combine with a tool such as ModelScope. RED exports models and model fragments to the PL file format.

3.1.1.3 Gourmand and other modelling tools

Gourmand is another thesis project of the Requirements, Models, Empirical SE group on IMM. It allows for extraction of models and model fragments as UML from Office tools such as Microsoft Powerpoint, Word and Visio.

All tools in Working Group “Requirements, Models, Empirical SE” including Gourmand will have the ability to export models in the PL file format. Furthermore, Gourmand will possibly also be able to export to XMI.

3.1.2 Users

This section describes speculations on potential users of ModelScope. The users presented here are based on the brainstorms and application scenarios of the domain analysis in section 2 and will then be summarized into a few concise roles.

3.1.2.1 Teachers

Teachers in courses that involves UML modelling, could find ModelScope interesting for analysing their student’s assignment models. They may for instance want to compare them to assess the average workload or use ModelScope as plagiarism detection.

Tasks:

- Compare sizes of delivered assignment models.
- Use various model metrics as indicators of plagiarism.

3.1.2.2 Model Researchers

Researchers working with models in any context. The obvious case is the study of UML models, but it may be interesting to investigate other types of abstract models that fits the structure.

Tasks:

- Model and Metric Exploration
- Bulk processing of models

- Extending the system with new probes and readings
- Extending the system with new visualisations
- Maintain and further develop ModelScope

3.1.2.3 Students

Students could be given the tool as part of modelling courses to analyse size metrics of their UML projects.

Tasks:

- Compare different versions of model over time
- Compare size metrics in relation to other students models.
- Export their results for use in assignments etc.

3.1.2.4 Other Thesis Students

Future thesis students might use ModelScope as part of their own research and analysis. It is entirely possible that other thesis projects might be derived from ModelScope. This may be extensions looking at other parts of models, for instance the visual representation of models and not just structural, or it may be something completely different, as long as the notion of probes and channels can be applied to it.

Tasks:

- Model Metric Exploration
- Develop ModelScope with new probes
- Develop new visualisations and channels
- Extend data models
- Create new parsers

3.1.2.5 Businesses

At some later point, the tool might be re-suited to operate in a business context. If for instance a project manager is asked to review a new version of the model, it could be very helpful to analyse it with a distance metric to see how much the model has changed since the last review.

Tasks:

- Follow the progress of a projects development over time
- Compare size metrics of project to previous projects
- Use metrics as indicator for the extend and cost of a project.
- Use distance metrics to assess the amount of change from one version to another.

3.1.2.6 Roles

The above described potential users fall in to two categories: those who uses the tool "as-is" and those who requires to be able to modify and extend it.

- Model Researcher
- Metric Developer

Note that the roles can overlap in some usage scenarios, e.g. the Model Researcher will often have to assume both roles to accommodate his work.

3.2 Goals and Qualities

Based on the context (section 3.1) and scenarios (section 2.1), the following two main goals have been derived:

- Exploring Metrics
- Exploring Models

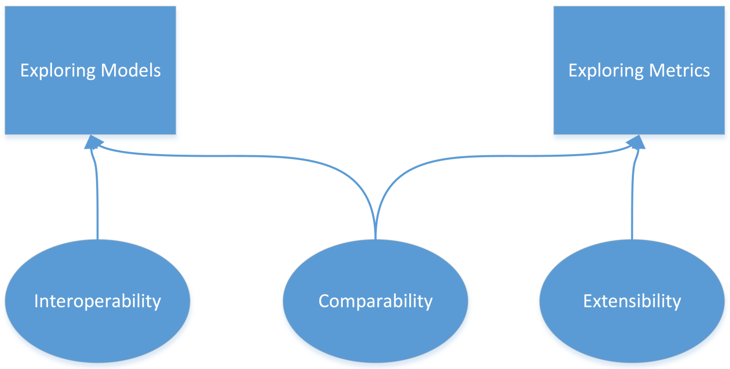


Figure 3.2: Overview of the most important goals and qualities

3.2.1 Exploring Metrics

This goal is not concerned with the actual discreet models, but only with how to measure them and compare these measurements. It is concerned with the process of finding and exploring good metrics for model sizes. It is inspired by the application scenarios of quality metrics in section 2.1.

This involves the following sub-goals and -qualities.

- Extensibility
  - Ability to implement new metrics
  - Ability to implement new visualisations
- Comparability (the visual and test result aspect)
  - Visualising metrics for comparisons

  - Comparing and testing metrics
  - Assistance in coming up with new metrics

3.2.2 Exploring Models

In contrast to “Exploring Metrics”, this goal is about investigating the actual model elements and exploring them using the defined metrics of ModelScope. In that sense, this goal is of a more static character, whereas “Exploring Metrics” is primarily about iteratively extending ModelScope. It can be derived from section 3.1.1 on page 17 on neighbouring systems and the application scenarios of section 2.1 on page 7 that ModelScope will have to be working with a range of different systems, and that it should not be constricted to those systems alone. Hence, what we could call “Interoperability” seems to be an important quality of “Exploring Models”. As exemplified in the scenario of section 2.1.1.1, “Comparability” is also a required quality of this goal.

Sub-goals and qualities of “Exploring Models”:

- Interoperability
  - Freedom in source of models
- Comparability
  - Visualising results
  - Using results outside ModelScope

Feature 1	**Alter settings of visualisations dynamically**
Description	Provide a way to alter settings of visualisations such as scaling, switching of axes etc. When using a visualisation to visualise metrics, it should be possible to change the visualisations settings while keeping the visualisation on screen to see the effects of the changes. Settings may include • Scale • Switch axes • Colours • Margins • Other visual properties

Table 3.1: Feature 1

Feature 2	**Analyse metrics of one model**
Description	It should be possible to import a single model and analyse its different metrics. The different metrics should be comparable within one model as different metrics could be compared across of models.

Table 3.2: Feature 2

3.3 Features

The features described in this section are derived from brainstorms on the scenarios of section 2.1, the neighbouring systems and users “Metric Developers” and “Model Researchers” of the Context analysis of section 3.1. This background illustrates the typical expected uses of ModelScope and should give an idea of what features are required for the final tool. Features are not in prioritized order.

Each use case is briefly described in the tables below:

Feature 3	**Analyse and compare metrics of two or more models**
Description	It should be possible to import a two or more models models and analyse their different metrics. The different metrics should be comparable across models but also within. I.e. it should be possible to look at the properties and metrics of all model elements as one mass regardless of which model they origin from.

Table 3.3: Feature 3

Feature 4	**Analyse and compare metrics for bulk set of models (i.e. load $>$ 50 models)**
Description	It should be possible to import a large set of models for examination without having to add each model manually and while maintaining a reasonable performance. Models of all supported formats should be imported in the same way. Once loaded they should be analysable as the use case in table 3.3.

Table 3.4: Feature 4

Feature 5	**Calculate metrics**
Description	It should be possible to quantify difference metrics, both direct and indirect. Metrics are defined as probes and readings.

Table 3.5: Feature 5

Feature 6	**Change visualisations dynamically**
Description	It should be possible to try out different visualisations dynamically while maintaining the setup of probes, channels and model elements such as only the visualisations change but the structure remains.

Table 3.6: Feature 6

Feature 7	**Export CSV**
Description	It should be possible to export the results of model probing to CSV for import in other tools such as Microsoft Excel.

Table 3.7: Feature 7

Feature 8	**Export PDF**
Description	It should be possible to export the graphical results of model probing and visualisations to PDF for use in report, research work, presentations etc.

Table 3.8: Feature 8

Feature 9	**Import from MagicDraw**
Description	It should be possible to import models created in MagicDraw in particular the file format MDXML which is a proprietary version of XMI.

Table 3.9: Feature 9

Feature 10	**Import RED and Gourmand models**
Description	It should be possible to import models extracted or created in tools such as RED and Gourmand which both uses the format PL.

Table 3.10: Feature 10

Feature 11	**Model Exploration like sketching**
Description	It should be possible for ModelScope to support the workflow sketched in the application scenario in section 2.1.1.1 especially Figure 2.4.

Table 3.11: Feature 11

Feature 12	**Scaling**
Description	It should be possible to scale visualisations to a user defined scale.

Table 3.12: Feature 12

Feature 13	**Visualise in a versatile range of charts**
Description	It should be possible to visualise probes and readings in a wide variety of visualisations. Examples: • Bar chart • Column chart • Pie chart • Line chart • Scatter plot • Gauge

Table 3.13: Feature 13

Feature 14	**Zoom**
Description	It should be possible to zoom in on selected parts of the visualisations as to reveal details otherwise hard to see. This may be when sizes varies greatly in size and the proportions makes it difficult to see the nuances in the smaller values. • Bar chart • Column chart • Pie chart • Line chart • Scatter plot • Gauge

Table 3.14: Feature 14

3.4 Information Models

The data models in this section are identified from the scenarios of section 2.1, the features of section 3.3 and brainstorming on these. The controls and outputs of an oscilloscope has been used as a metaphor and the class names and structures described here are heavily inspired by this analogy. The actual implementation is modelled after the diagrams shown here, but does deviate in certain places. An overview can be seen in Figure 3.3.

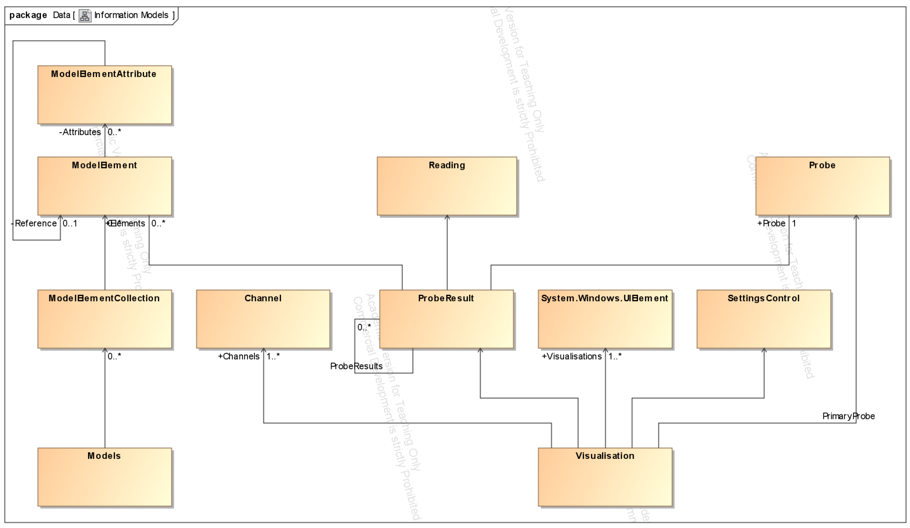


Figure 3.3: Overview of classes and associations

3.4.1 Models and Model Elements

The data structure of models and their elements can be seen in Figure 3.4.

A model is represented as a `ModelElementCollection` and holds a number of `ModelElements` which in turn can have any number of `Attributes`, some of which may be references to other `ModelElements`. This rather simple and flat data structure is perfectly able to hold the necessary data and using LINQ[1] it is easy to query and output the flat structure as tree-like temporary virtual structures when needed.

[1].NET declarative query syntax

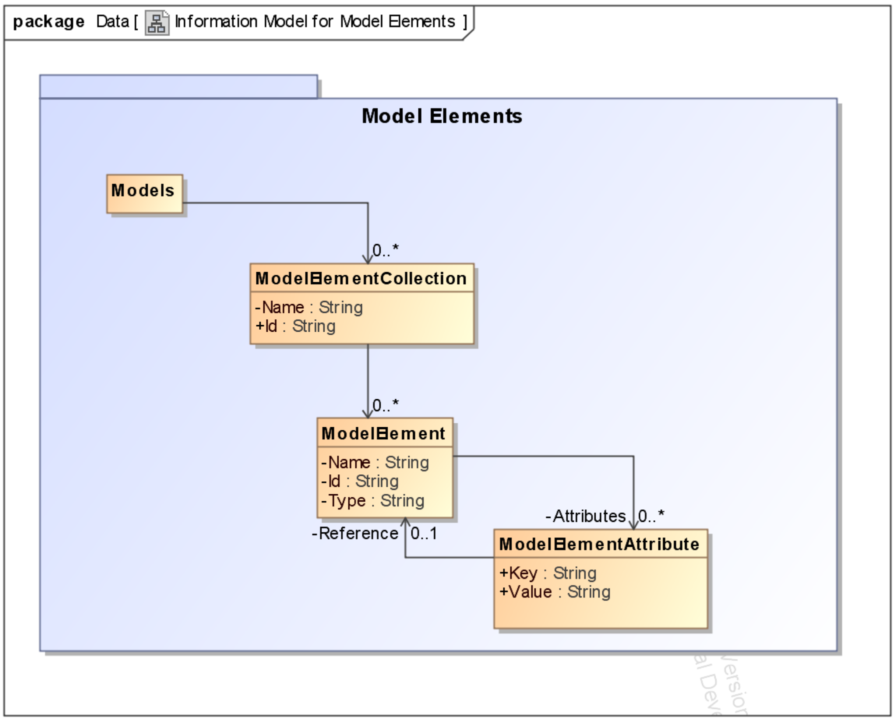


Figure 3.4: Class diagram of the data structure for models and model elements

3.4.2 Visualisations

The data structures of visualisations can be seen in Figure 3.5. The visualisations are visual representations of probe results in any way. Typically it would be a bar chart, pie chart or something similar, but it could also be graphs, tables or text in different font sizes. A visualisation results in a list of `UIElements`, which is just any kind of graphical element.

The visualisation may have some general settings available to the user. These can be attached in the form of a `SettingsControl` which will contain both the UI part and the data part of the settings. Settings could be colours, change of axes, scales etc.

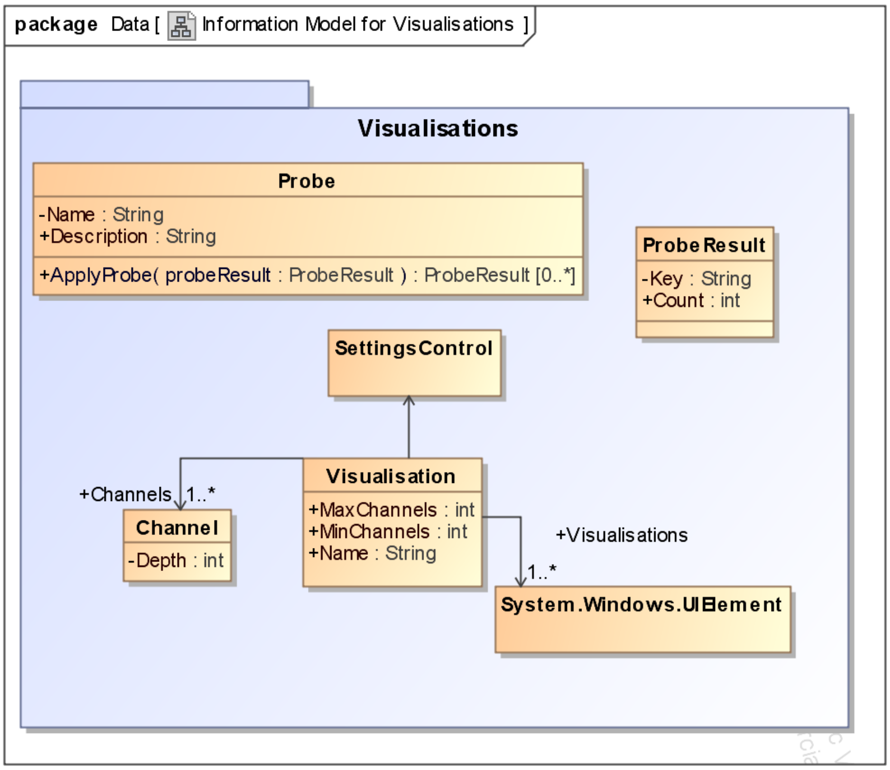


Figure 3.5: Class diagram for visualisations

A visualisation will show measurements in `Channels`. Figure 3.6 shows an example of a visualisation in the form of a column chart with two output channels: R1 and R2. The primary probe shows as the first grouping on the X axis i.e.

"A" and "B". The channel R1 is just any reading of that probe and the result is the height of the column on the Y axis. The channel R2 has a depth of two and holds a secondary probe which splits it into R2.1 and R2.2. The results of that probe is displayed as a stacked column with the reading as height on axis Y. The readings of a channel with depth two does not have to be stacked, but it simplifies the distinction of channels in the diagram.

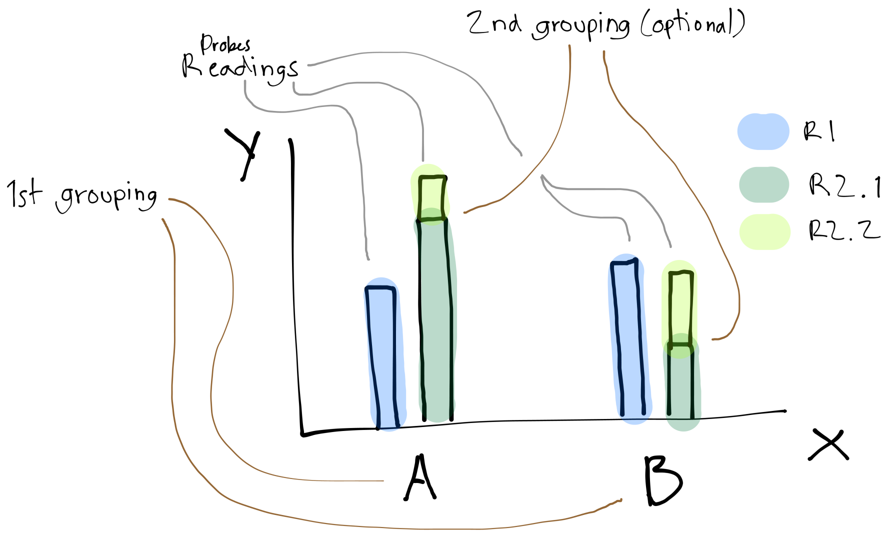


Figure 3.6: A sketch of how probe results can be grouped and displayed in a column chart. This chart contains two channels: R1 and R2. R2.1 and R2.2 are both part of channel R2, even though it contains another probe. I.e. R1 is of depth one R2 is of depth two.

As Figure 3.6 shows, a column charts can have a minimum of one channels and a maximum of ∞ (though that may clutter the readability of the diagram somewhat). The channels has a maximum depth of two. A pie chart as in Figure 3.7 can only have one channel with depth one i.e. only one probe can be applied and to only one channel. It can however be repeated to show multiple channels.

When thinking up a new visualisation, these are considerations to make before the implementation:

- Minimum number of channels (some may require more than one channel to function properly e.g. a scatter plot)
- Maximum number of channels

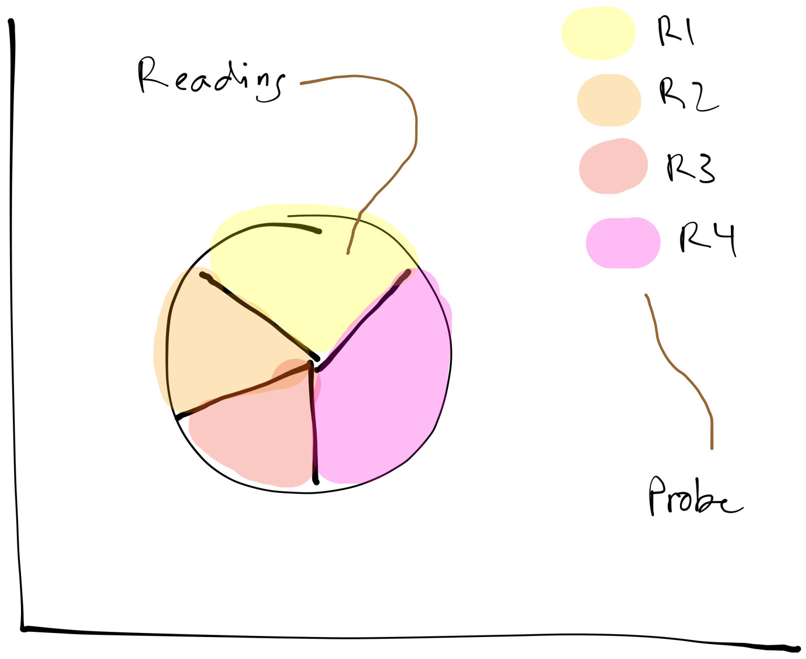


Figure 3.7: A pie chart with only one channel of depth one i.e. only one probe. The probe determines the grouping and the reading is displayed as percentage of the pie.

- Depth of channels
- What kind of settings and variations of the visualisation should be available to the user.
- Can the maximum number of channels be expanded by repetition of the visualisation.

3.4.3 Probes

Figure 3.8 shows the information model for probes and probe results. A probe is applied to a set of model elements and will itself result in a list of probe results which each holds a set of model elements. An illustration of this process can be seen in Figure 3.9 which is inspired by the scenario in section 2.1.1.1.

Note that the probe results does not need to be subsets of their parent nodes in a strict sense. Probes can result in any set of model elements; larger or smaller than their parents. A `Reading` is a quantifiable measure of a `ProbeResult` and can thus be retrieved from any level of the tree in Figure 3.9.

Examples of readings could be:

- Count the number of elements
- Count the number of references
- Count the number of attributes
- The average number of references pr. attribute
- Any indirect measure calculated on the basis of the probe result.

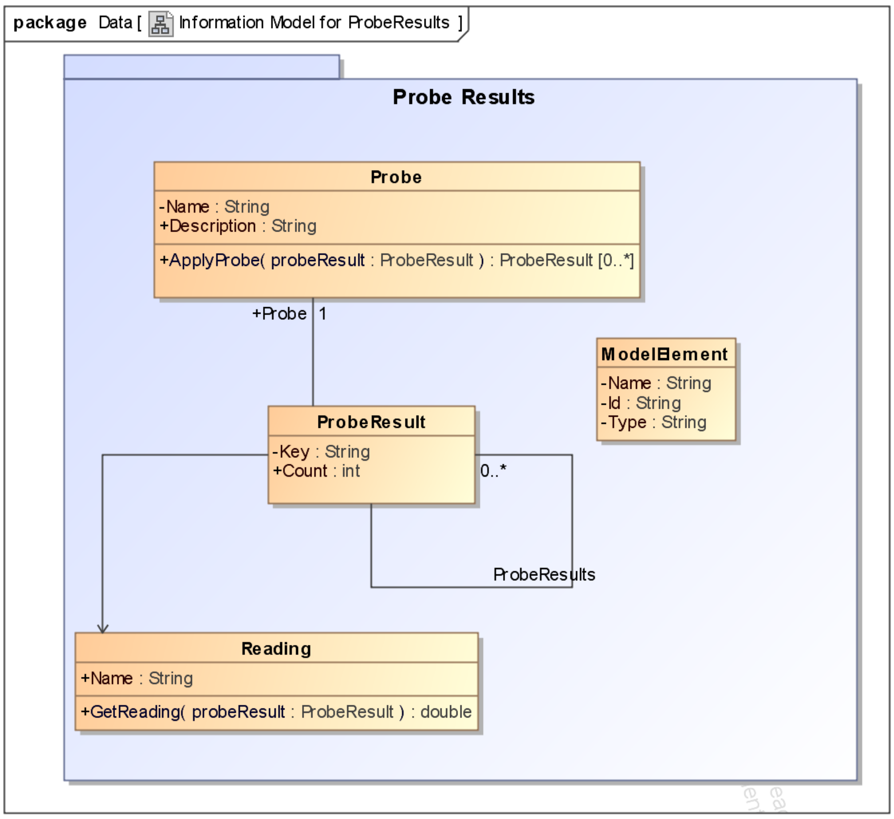


Figure 3.8: Class diagram of Probes and ProbeResults

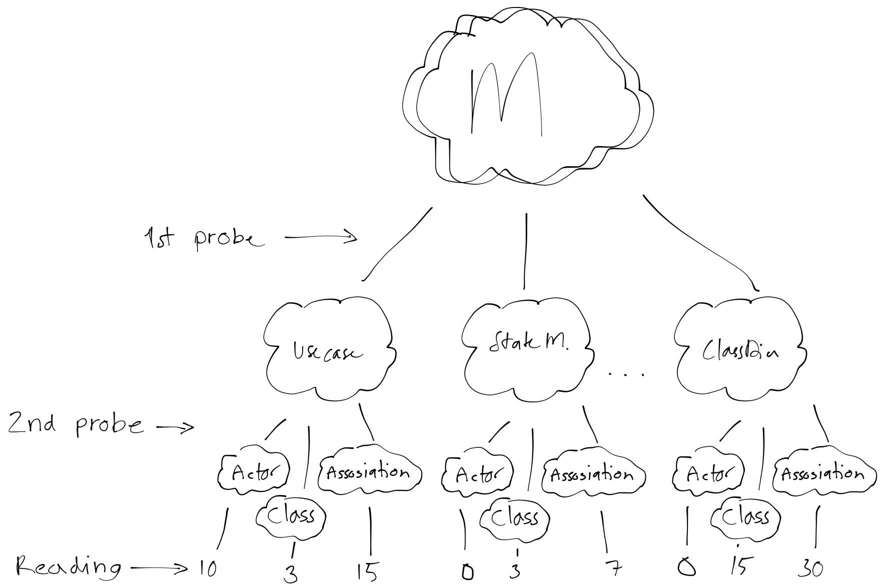


Figure 3.9: Illustration of how probes are applied to sets of model elements. The height of the tree constitutes the number of probes and also the depth needed of the channels to display the readings. Readings can be made at any level of the tree. In the illustration the readings are displayed as integers in the bottom leafs.

CHAPTER 4

Design

4.1 User Interface

Our metaphor for the ModelScope project is the oscilloscope and that also serves as a basis for the initial sketches of the user interface. The idea is to have a tangible user interface that encourages exploration of the different probes and visualisations as exemplified in 2.1.1.2.

Hand-drawn sketches has been used for initial much-ups. Microsoft Blend has been used for prototyping and testing prototypes.

The first prototypes were wizard-based. I.e. they directed the user through a number of screens beginning with the selection of models to be imported, on to choose which probes to use (see Figure 4.1) and ended in an interface with visualisations displayed along with oscilloscope-like options.

The sketch on Figure 4.2 shows some of the components identified in the information models of section 3.4. The screenshot of Figure 4.3 shows an early screenshot of ModelScope with an actual implementation. Some of the elements has changed places but the essentials are still there: possibility of changing visualisation type, selection of probes and channels to the left, the visualisation to the right and a settings panel for altering settings regarding the selected vi-

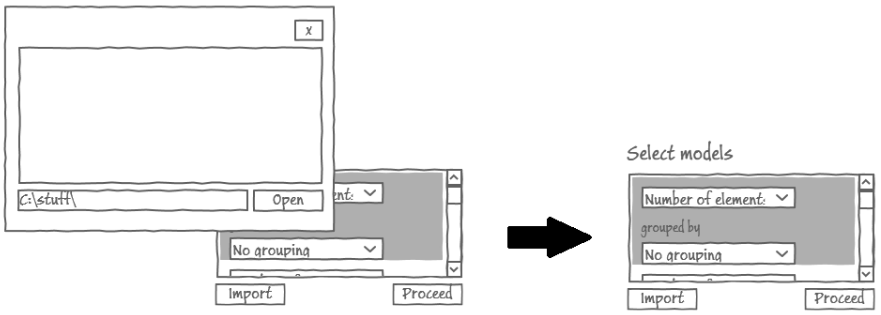


Figure 4.1: The initial screen in the first prototype

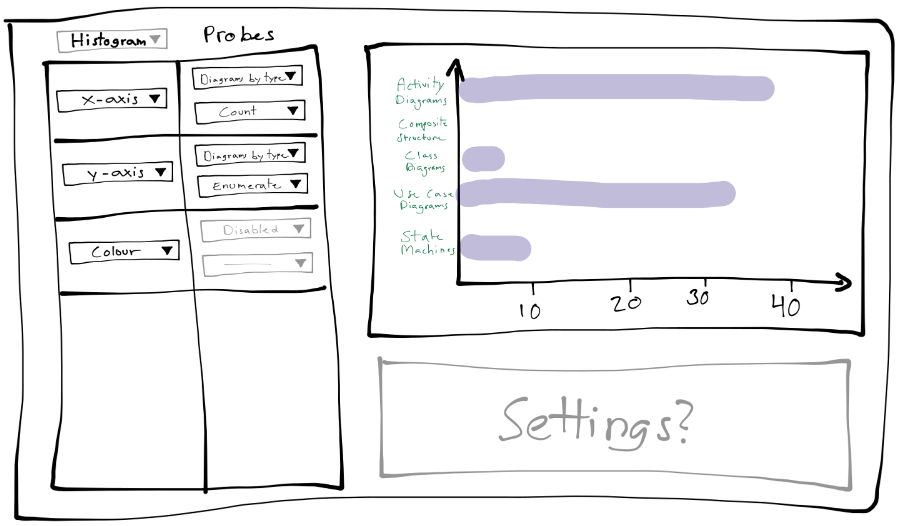
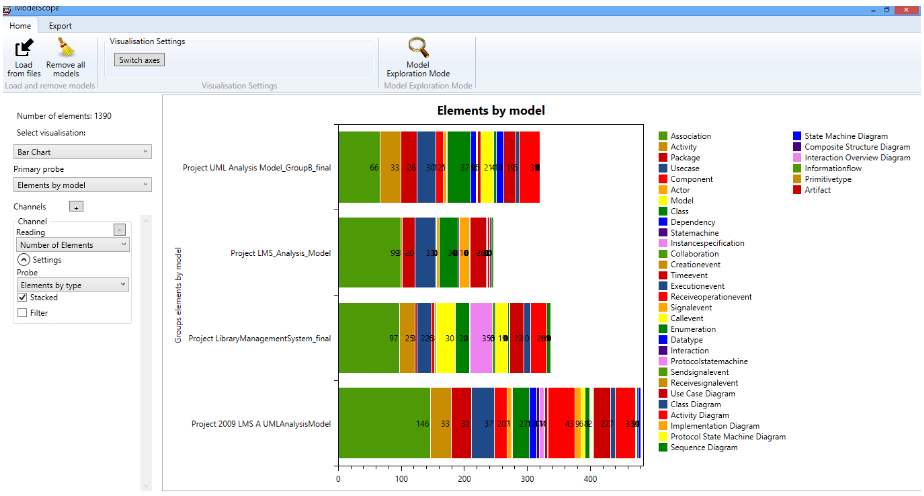


Figure 4.2: One of the initial hand drawn sketches of a possible UI for ModelScope

sualisation. The wizard-idea was scrapped during prototyping to give the user the ability to import or remove new models gradually while working with the visualisations.



Number of elements: 1390
Select visualisation: Bar Chart
Primary probe: Elements by model
Channels
Channel Reading: Number of Elements
Settings
Probe: Elements by type
[✓] Stacked
[ ] Filter

Figure 4.3: An early screenshot of ModelScope with a stacked bar chart as visualisation

4.2 Architecture

The overall internal structure of ModelScope can be seen in the diagram of Figure 4.4.

Firstly, the design choice of building a stand-alone application and not a plugin for existing software was based on the goal of "Interoperability" identified in section 3.2.

One of the other main concerns of the architecture is the goal of section 3.2; "Extensibility". Modularity has therefore been a priority and it is especially important that `Probes`, `Parsers` and `Visualisations` are exchangeable and extendible.

The architecture of the individual modules follows that of the described information models in section 3.4 and other details about implementation are provided in section 5.

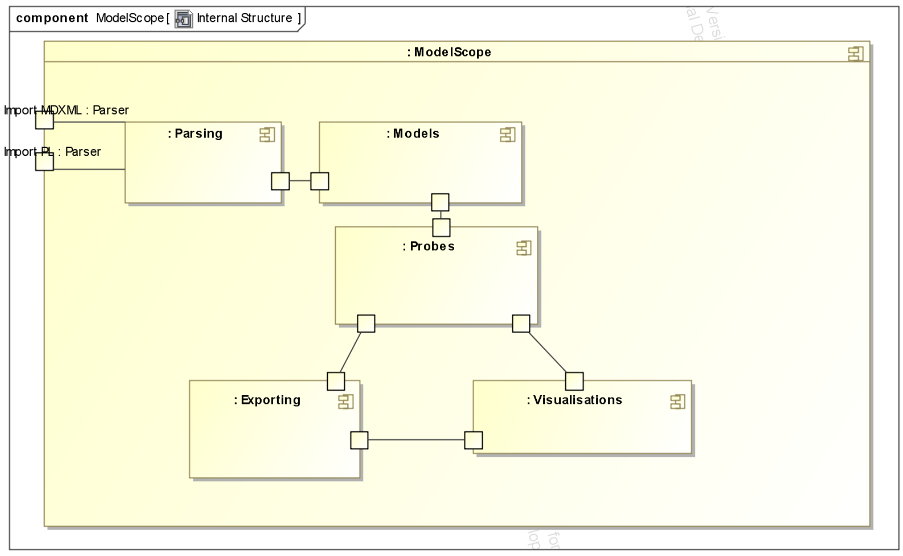


Figure 4.4: Composite-Structure Diagram of the internal structure of ModelScope

Chapter 5

Implementation

5.1 Technology

5.1.1 Platform and programming languages

The technological basis for the development of ModelScope is Microsoft Visual C# and the .NET platform. The IDE used is Microsoft Visual Studio 2012. Visual Studio provides a range of helpful features including code completion, 3rd-party plug-in handling and updating, test-facilities and source control with integrated task management. C# was chosen as primary programming language because of familiarity and availability of help and online documentation.

Because of the choice of platform, ModelScope will currently only operate on the Microsoft Windows platform. One possibility of cross-platform behaviour could be achieved by using a remote-desktop set-up (e.g. Citrix) to allow clients on other operating systems to connect. Another possibility is to port ModelScope to Linux using cross-platform open-source development frameworks such as Mono. Ports to other operating systems has not been considered further.

5.1.2 Version control

Visual Studio 2012 provides for several options with regards to version control, most notably Team Foundation Server and GIT. In this case, Team Foundation Server has been chosen for versioning. Version control with Team Foundation Server provides options for branching which can be extremely useful for making changes to large parts of the code, without having to worry about role-back. Another useful feature of Team Foundation Server is the close connection with task-management which can be operated both online and from within Visual Studio. This makes it more manageable to track changes to the code over time.

5.2 Libraries and Components

Microsoft .NET 4.5 includes libraries and components in abundance and ModelScope takes advantage of these wherever possible. This goes for data structures like enumerable `List<T>` and hash maps like `Dictionary<key,value>` as well as graphical frameworks as Windows Presentation Foundation and all the components that follows this suite.

There are not as many free 3rd party libraries for .NET as is the case with for instance Java. The chosen libraries for ModelScope are picked according to functionality, but also with license in mind. Open-Source has been considered a plus as it opens up for the possibility of changing or expanding the libraries as well to fit ModelScope's need in future work.

Visual Studio 2012 (and later editions) has a feature called "Nuget Packages" which is essentially a feature that resembles that of a package manager in Unix-systems. It allows developers of .NET libraries and components to make them available through a package managing interface (a console) from within the IDE. The "Nuget" system then handles dependencies and updates of the libraries.

All libraries used in ModelScope utilizes this "Nuget" functionality which makes it very easy to make sure all libraries are up to date and dependencies are taken care of. This is important, since because of the goal "Extensibility" (see section 3.2), it must be assumed that Metric Developers will be developing and updating the code of ModelScope regularly.

5.2.1 Libraries for Visualisations

It is of course possible to implement visualisations manually, and it may even be a good idea if they are simple enough. That could for instance be text-based visualisations as tables or just numbers of various types. But as soon as it gets more complicated and graphical than that it may be a good idea to be able to rely on a library constructed for the purpose.

The main requirements for visualisation libraries were:

- Open license (preferably open source) with active development
- Variety of visualisations within the same library, for consistent style as well as ease of implementation
- Has to have at least these basic diagram types:
  - Bar Chart
  - Column Chart
  - Pie Chart
  - Line Chart

5.2.1.1 Investigated Libraries

Below are the libraries considered for ModelScope. Only free libraries have been considered. Note that the architecture allows for use of multiple libraries for visualisations. Each visualisation could use its own library but this particular implementation will rely on the same library for all visualisations.

WPF Toolkit Data Visualization The WPF Toolkit was originally made as a supplement to the Microsoft WPF framework, which at that time was scarce of controls and components. The toolkit has visualisation part that fits ModelScope's requirements and the license is open as well. Since Microsoft WPF has grown substantially since its release, WPF Toolkit has not been updated since February 2010.

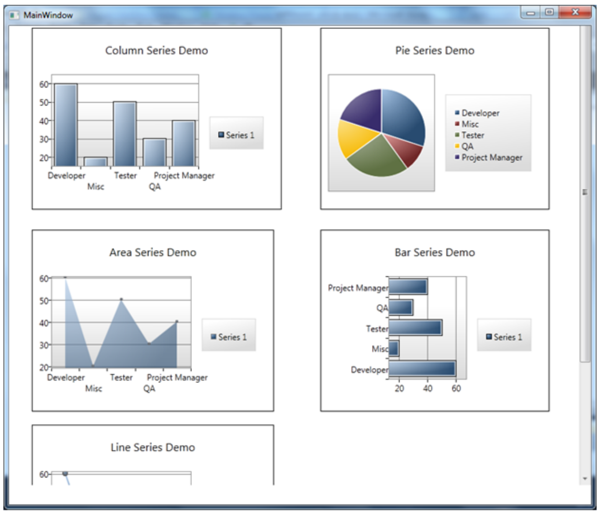


Figure 5.1: Some of the visualisations available in WPF Toolkit

WPF Chart Control With Pan, Zoom and More [1] This library is a part of a WPF tutorial. It provides a lot of details but only includes one finished type of visualisation. It is unsupported but public domain licensed.

OxyPlot OxyPlot is by far the most comprehensible library examined, and contains a vast number of visualisations.[2] It is Open-Source and open-licensed. It is currently in active development with the last stable version being from January 2014.

Modern UI (Metro) Charts for Windows 8, WPF, Silverlight Modern UI Charts is an open-source library for WPF in the style of Windows 8 Apps (formerly known as "Metro"). It is visually very appealing (see Figure 5.2) and contains five different types of charts.

[1]From http://www.codeproject.com/Articles/17097/WPF-Chart-Control-With-Pan-Zoom-and-More

[2]See http://www.objo.net/oxyplot/ExampleBrowser/ for a demonstration of all available visualisations.

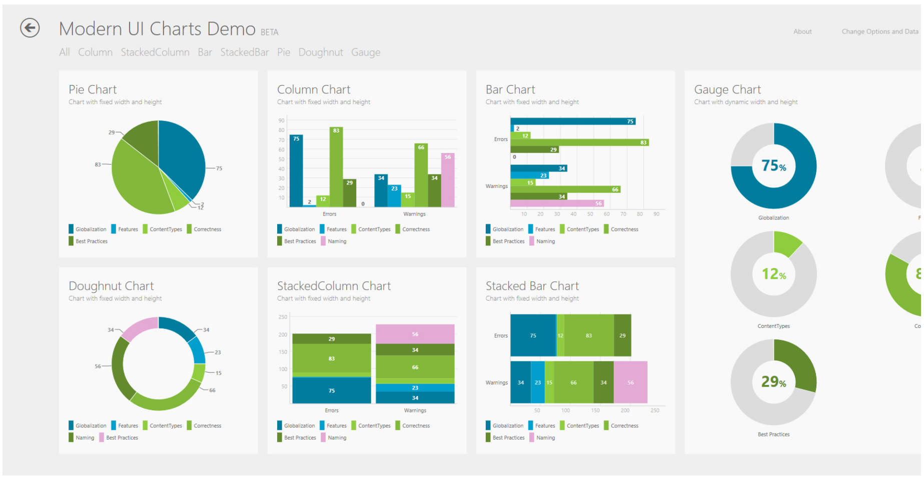


Figure 5.2: Examples of Modern UI Charts

5.2.1.2 Choice of Primary Visualisation Library

OxyPlot has been chosen for the primary visualisation library. It has by far the greatest number of different visualisations and this will make it easier to implement new visualisations into ModelScope with code of similar syntax. It will also make the visualisations more even in visual appearance.

The choice of OxyPlot as primary library does not limit the use of other libraries in visualisations.

5.2.2 Libraries for Parsing

As can be seen in section 5.3, the only two parsers implemented are for XMI (MDXML) and PL.

Microsoft .NET has a comprehensive build-in library (`LINQ-to-XML`) for handling XML structures via the declarative syntax of LINQ, and this library is utilized excessively when parsing XMI.

The very simple structure of the PL files are dissected through the use of .NET's build-in Regular Expression library.

5.2.3 Libraries for Exporting

Exporting is divided into two parts:

- Exporting visualisations
- Exporting data of visualisation

Exploring visualisations uses OxyPlot's PDF Report Generator Library. This means that visualisations using other libraries than OxyPlot for visual output, would have to implement their own export functionality as well.

Exporting data of visualisation uses the open-source library `CSVHelper`. It currently exports in up-till-two dimensions with any number of channels.

5.2.4 Libraries for Graph Visualisation ("Exploration Mode")

The library used for "Model Exploration Mode" is called `GraphSharp` and is an open-source .NET library. It allows for other structures than trees to be visualised as well, and is generally very flexible. An illustration of "Model Exploration Mode" using a graph algorithm instead of the default tree algorithm (see Figure 6.5 on page 59) can be seen in Figure 5.3.

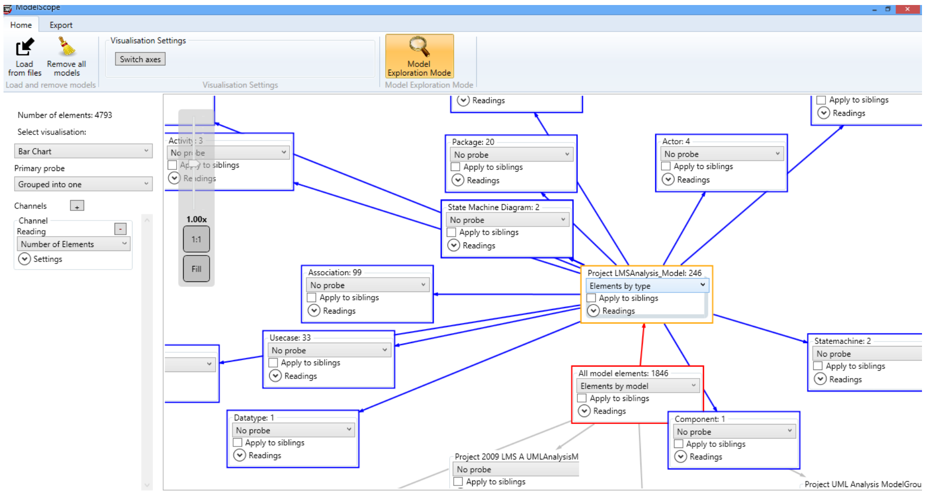


Figure 5.3: "Model Exploration Mode" displayed with GraphSharp library using a graph-visualisation. Zooming and panning is of course possible.

5.3 Parsing

Parsers are created by implementing the ModelScope.Parsing.Parser interface. The method Parse() returns a ModelElementCollection with the parsed models and model elements.

5.3.1 Parsing MagicDraw files

MagicDraw stores its models in the MDXML file format. MDXML files from MagicDraw contains a vendor specific XMI document in which all model data is stored.

XML Metadata Interchange (XMI) is a standard for exchanging meta-data information via XML. It is widely used for storing models such as UML. MagicDraw uses XMI for this specific purpose, and this parser implementation is assuming the XMI-document to hold just UML models. This implementation handles output from MagicDraw version 16.9. It will parse other versions as well, but be aware that there could be version specific features which will not be accommodated.

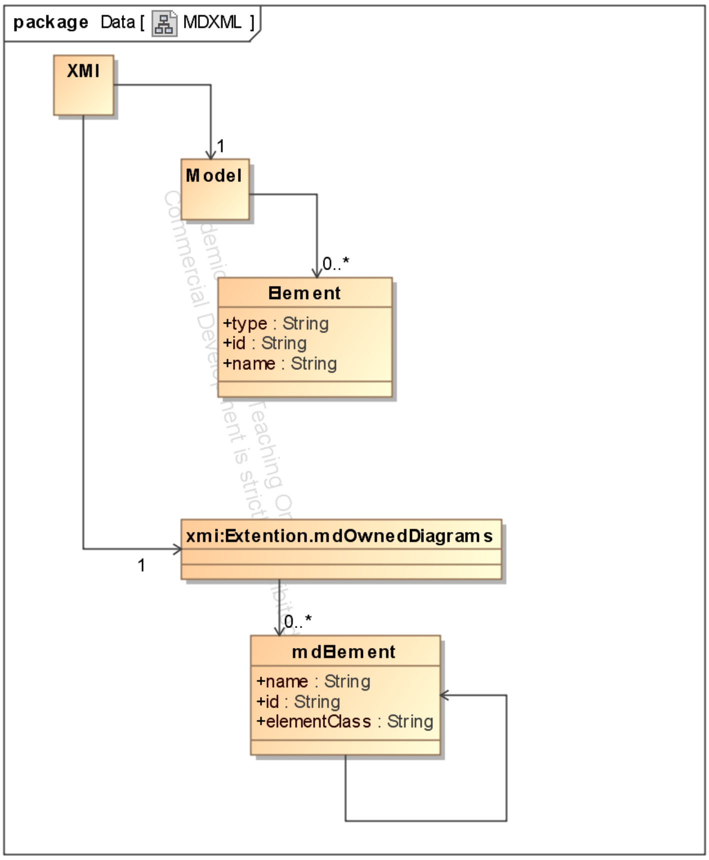


Figure 5.4: An illustration of the relevant parts of the XMI structure of an MDXML document

5.3.1.1 Identifying relevant parts of MDXML document

The MDXML document contains vast amounts of information about the MagicDraw UML model, not all of those relevant to ModelScope. Since ModelScope focuses on structure, all information regarding the visual composition of the models is discarded.

The main interesting part of the MDXML document is the *xmi.XMI.uml:Model* node which contains the primary model elements, associations and properties.

Example of `packagedElement`:

```
<packagedElement xmi:type='uml:Model' xmi:id='
    _16_9_877027b_1351843403926_946141_2123' name='eMenu'
    visibility='public'>
       <packagedElement xmi:type='uml:Use Case' xmi:id='
           _16_9_877027b_1351843554050_694351_2197' name='list
           items' visibility='public'/>
</packagedElement>
```

Model elements are stored in `<packagedElement>` nodes which has the following structure: Attributes:

- xmi:type='uml:[Type]'
- xmi:id='[String ID]'
- name='[Name of model element]'
- visibility='public'

References are done as nested child elements of `<packagedElement>` or in `<ownedAttribute>` with the following attributes:

- xmi:type='uml:[Property]'
- xmi:id
- type='[ID of referenced model element]'
- association='[ID of association element]'

5.3.1.2 Diagrams

Diagrams are handled substantially differently than other model elements in MDXML. As can be seen in Figure 5.4, diagrams reside in "`<mdOwnedDiagrams>`" which is a part of a proprietary MagicDraw extension to XMI. "`<mdElements>`" have children that refers to IDś of model elements, and this is how it is determined which model elements are held by a diagram.

When parsing diagrams to ModelScope's data model, the diagram itself is translated to a `ModelElement` and all model elements contained within that diagram will be linked to the diagram via `Attributes` with `References`.

5.3.2 Parsing PL files

PL files contain a prolog data structure describing a model. Each line contains a model element (`me(...)`) with various attributes and references.

Example of PL file content:

```
:-module('Sample1',[]).
me(model-0,[annotation-id(1),ownedMember-ids([1,2,3]),name-'Data'
    ,visibility- (public)]).
me(comment-1,[body-'Author:Jacob.\nCreated:08-11-13 18:31.\nTitle
    :.\nComment:.\n',annotatedElement-id(0)]).
me(package-2,[referentType-'Package',referentPath-'UML Standard
    Profile',href-'UML_Standard_Profile.xml#
    magicdraw_uml_standard_profile_v_0001']).
me(package-3,[ownedMember-ids([4,5,8]),name-'Classes',visibility-
     (public)]).
```

Each line is interpreted as a `ModelElement` and `ownedMember-ids` are used for references.

Note that PL does not currently support diagrams.

5.4 Test

The implementation has been subjected to regular unit testing via Visual Studios Test Suite.

UI functionality is not tested automatically.

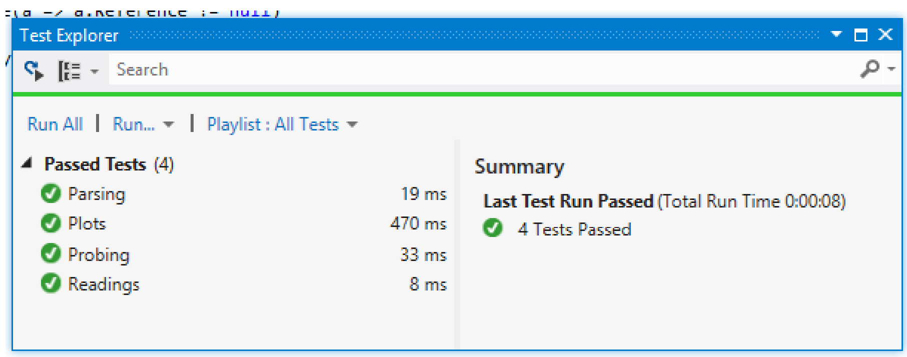
Unit Testing is divided into four areas of the code:

- Parsing
- Plots
- Probing
- Readings

All tests involving calculations and manipulation with a model set, uses the function `SetupTestModels()` to ensure a uniform setup.

5.4.1 Results of Unit Testing

All test completes successfully.



```
Passed Tests (4)
  Parsing     19 ms
  Plots      470 ms
  Probing     33 ms
  Readings     8 ms

Summary
Last Test Run Passed (Total Run Time 0:00:08)
4 Tests Passed
```

Figure 5.5: Test results from Visual Studio

5.4.2 Code Coverage

According to Visual Studio the unit test code coverage is **51.33 %**. What this means is that 51.33 % is reached through testing; not the 51.33 % of the functions are tested rigorously.

5.4.3 Performance

The performance of ModelScope has been tested manually to assess responsiveness of the GUI. Using ModelScope with < 5000 model elements seams completely fluent, so a stress test was devised. The test was conducted as follows: First, the parser was modified to import all given files 10 times instead of one. Then 11 models with a collective size of approximately 25 MB where loaded into ModelScope, resulting in a total of 111 models with 250 MB of model data, 66390 model elements and 129980 attributes.

The result was that ModelScope responded with a significant lag in responsiveness; 0.5-1.0 seconds whenever changing the visualisation either by probes or visualisation-settings. The lag did not in any way make the tool unusable, however, and the amount of model elements treated in this test must be considered unusual.

It is therefore concluded that ModelScope does satisfy the quality of performance in bulk computation of section 3.3.

Chapter 6

Validation

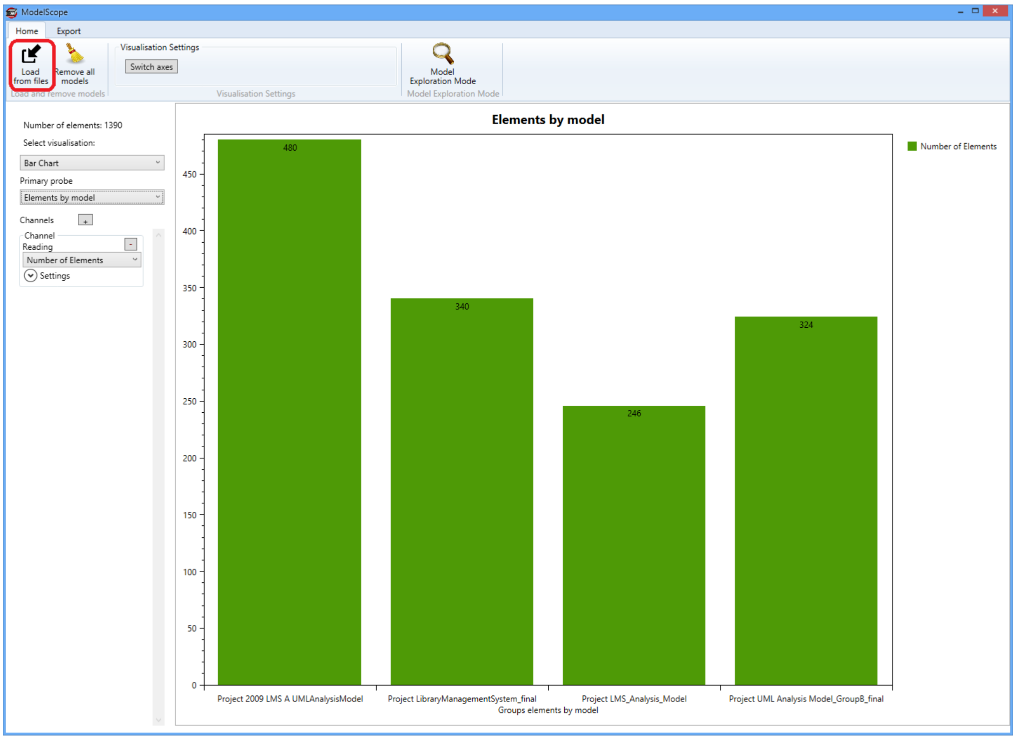
In this section the different application scenarios described in section 2.1 are examined again but this time with respect to the tool support of ModelScope.

6.1 Model Research

This section re-examines the application scenario of section 2.1.1.1. In the scenario, four models are analysed for similarities in structure using manual processing and visualising. Using ModelScope for tool support, the case is here revisited.

The four models are created with MagicDraw and are therefore in the MDXML format. The MDXML files can be imported directly into ModelScope as can be seen in Figure 6.1 which also shows an initial visualisation of the number of model elements pr. model. ModelScope parses the files at once into its internal data structure on which all calculations are performed.

This gives a simple overview of how the models compare in size (i.e. number of model elements). Following the path of scenario 2.1.1.1 we can choose to apply another probe to each of the models. This is done by selecting a probe in


Number of elements: 1390

Select visualisation: Bar Chart

Primary probe: Elements by model

Channels

Channel

Reading: Number of Elements

Figure 6.1: All four models imported to ModelScope. Initial screen shows diagram of number of elements grouped by model

the “Settings” panel of the channel. The result of the procedure can be seen in Figure 6.2.

To inspect part of the diagram in more detail, zooming and scaling can be applied as exemplified in Figure 6.3. This is achieved simply by holding down the CTRL-key and dragging over the part of the diagram as it is partly done in Figure 6.2.

Replicating the results of Figure 2.2 in the application scenario can be done by simply switching the probes and the axes. Figure 6.4 gives an example of this. Note that the results does not match completely as different methods of measurements was used. See section 5.3 for information on how the models are parsed.

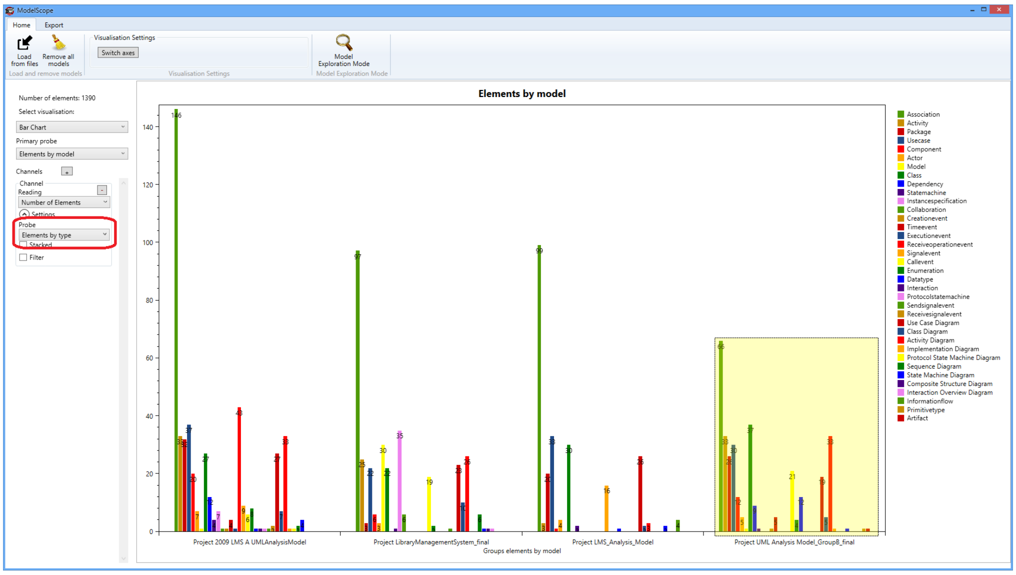
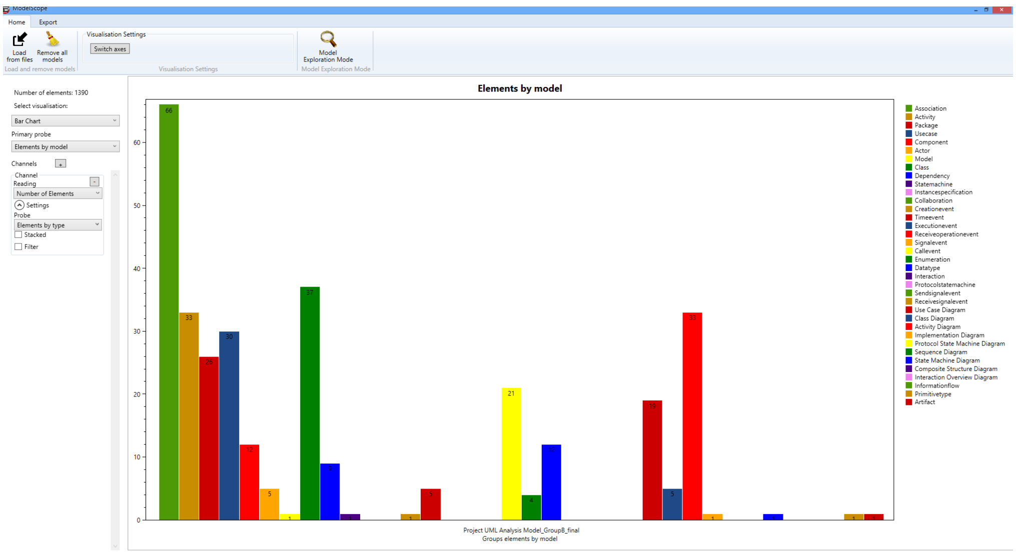

Number of elements: 1390
Select visualisation: Bar Chart
Primary probe: Elements by model
Channel Reading: Number of Elements
Probe: Elements by type
Stacked
Filter

Figure 6.2: Number of model elements grouped by type. Yellow area indicates intended zoom-section. Figure 6.3 shows the result of the zooming.

If we wanted to replicate the table views we could either use the export functionality of ModelScope and export to CSV and then into a spreadsheet tool, or we could write a simple extension visualisation to ModelScope that would display data in a table instead of a chart.

If ModelScope does not support a certain probe or reading it can be added as an extension. In [Sto10] the “number of elements without a name” is used as a metric. All that is necessary to do to add such a new reading to ModelScope is to


Number of elements: 1390

Select visualisation: Bar Chart

Primary probe: Elements by model

Channel

Reading: Number of Elements

Probe: Elements by type

Stacked ☐

Filter ☐

Figure 6.3: Zoom on group B's model. Note that we are still looking at the same diagram but zoomed to a part of it. Scaling adjusts automatically.

implement the interface `Reading` and add the class to `Readings.ReadingList`:

```
public class NumberOfElementsWithoutName : Reading
{
   public override string Name
   {
      get { return "Number of elements without name"; }
   }

   public override double GetReading(ProbeResult probeResult)
   {
      return probeResult.Elements.Where(element => element.Name ==
           "").Count();
   }
}

...

AddReading(new NumberOfElementsWithoutName());
```

Note how the use of declarative LINQ simplifies the syntax and makes it easy

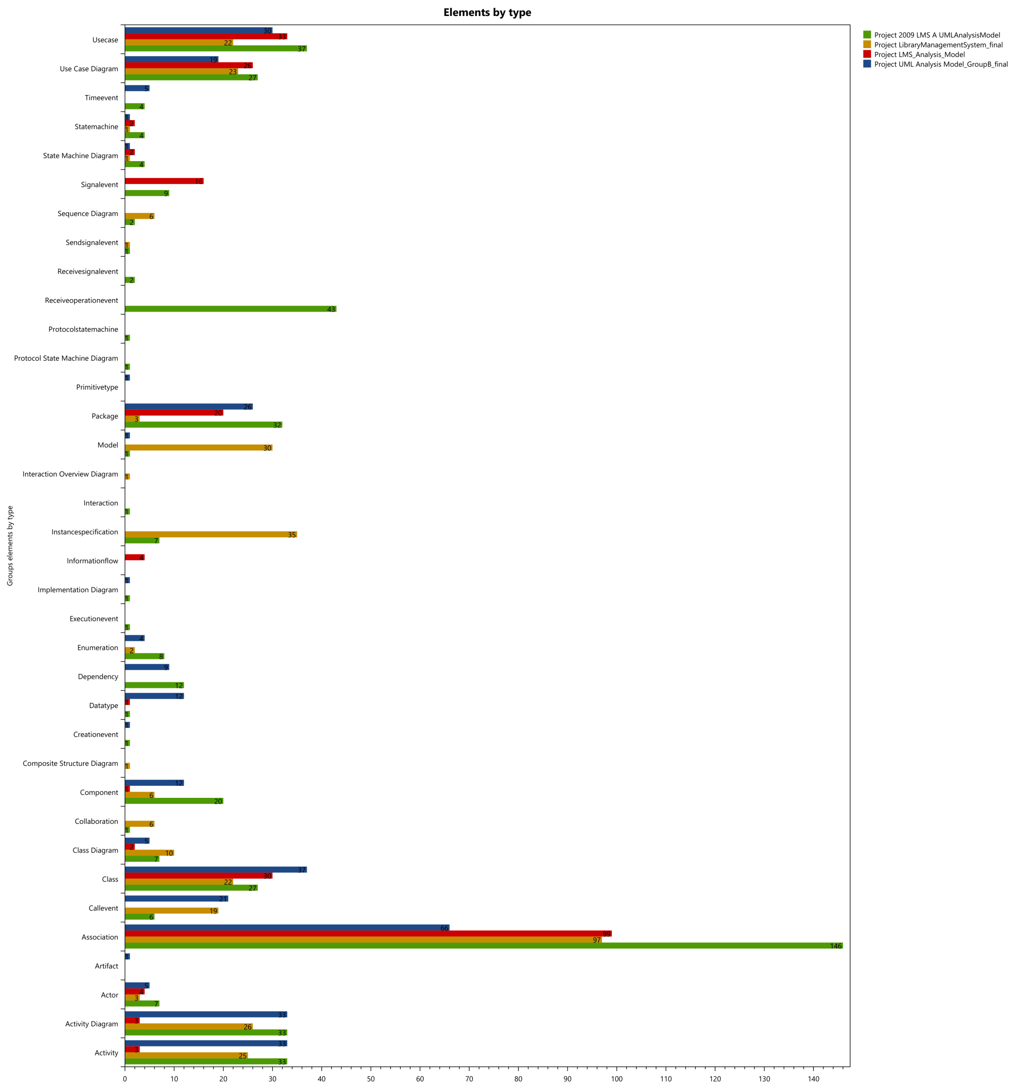


Figure 6.4: Replication of Figure 2.2 using ModelScope. The Figure has been exported from ModelScope using the "Export PDF" feature.

to both code and comprehend: `...Where(element => element.Name == "")`.

6.2 Metric Exploration

This section re-examines the scenario of section 2.1.1.2 about exploration of metrics. Section 2.1.1.2 speculates on a model researcher trying out different types of probes and readings in an exploratory way. Here we try to utilize ModelScope for the processes described in the scenario.

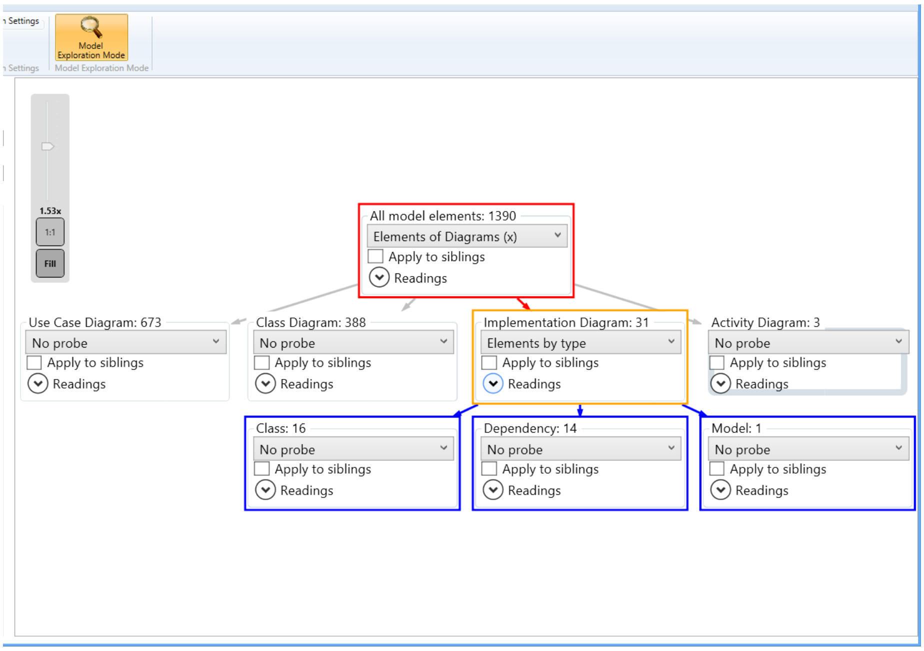
Firstly some models would be imported. This could be many or few and from different sources. It could also be fragments of models extracted via Gourmand or from RED using the PL format. Once loaded into ModelScope, there are no difference between the models with regards to their origin; only the abstract structure is preserved.

"Model Exploration Mode" enables the researcher to explore the models element in a fashion much like the sketch of Figure 2.3 on page 13. This can be seen in Figure 6.5 where Model Exploration Mode is used to dynamically apply probes in a nested way, just like it was done by hand in Figure 2.3. This way of exploring the model data has shown in this study to be a good way to inspire new probes and readings.

If Model Exploration Mode inspires the researcher to probe in a way not yet supported by ModelScope, the researcher can implement the probe as an extension. This is done by making a new class implementing the interface `Probe` and adding the probe to `Probes.ProbeList`

```
public class ElementsOfDiagrams : Probe
{
   public override string Name
   {
      get { return "Elements of Diagrams"; }
   }

   public override string Description
   {
      get
      {
         return "Child-elements of diagrams, by diagram-type";
      }
   }
```



All model elements: 1390
Elements of Diagrams (x)
Apply to siblings
Readings

Use Case Diagram: 673
No probe
Apply to siblings
Readings

Class Diagram: 388
No probe
Apply to siblings
Readings

Implementation Diagram: 31
Elements by type
Apply to siblings
Readings

Activity Diagram: 3
No probe
Apply to siblings
Readings

Class: 16
No probe
Apply to siblings
Readings

Dependency: 14
No probe
Apply to siblings
Readings

Model: 1
No probe
Apply to siblings
Readings

Figure 6.5: Model Exploration Mode resembling the work flow of Figure 2.3

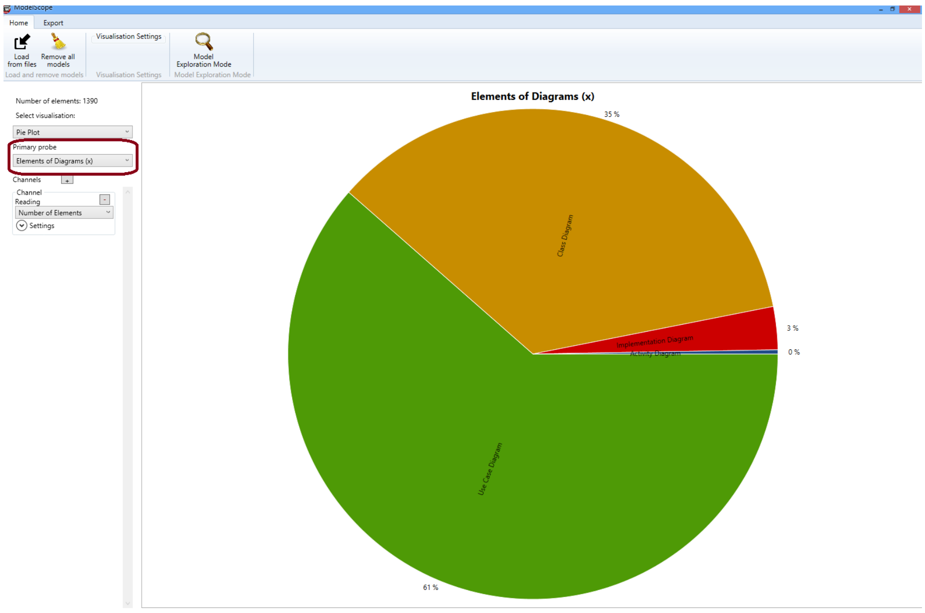
```
  public override List<ProbeResult> ApplyProbe(ProbeResult
      probeResult)
  {
    var results = from e in probeResult.Elements
                  where e.Type.ToLower().IndexOf("diagram") > -1
                  from a in e.Attributes.Where(a => a.Reference !=
                      null)
                  group a.Reference by e.Type into refs
                  select new ProbeResult { Elements = refs.ToList()
                      , Key = refs.Key };

    return results.ToList();
  }
}
```

See appendix A for a more hands-on manual on how to extend ModelScope.

`ApplyProbe` in the above example utilizes the declarative LINQ to find the results to return, but an imperative solution would work just fine as long as it returns a list of `ProbeResult`. The code finds all model elements with the string “diagram” in its type and returns the collective set of child elements (references of the diagrams) grouped by type of diagram. Figure 6.5 gives an illustration of its application.

When a probe has been implemented it should be tested and visualised. After rebuilding the code of ModelScope we can again load in models and find the newly implemented probe in the probe-lists. An example of a visualisation can be seen in Figure 6.6



Number of elements: 1390

Select visualisation: Pie Plot

Primary probe: Elements of Diagrams (x)

Reading: Number of Elements

Figure 6.6: Visualisation of the newly implemented probe

Chapter 7

Conclusion

This section summarises our achievements and results, discusses how they satisfy the original vision and highlights possible future directions.

7.1 Summary and Contributions

This thesis has analysed different application scenarios for exploring software model metrics (section 2.1), has created a tool accommodating them (sections 2-5), and has validated the tool against the application scenarios (section 6).

Using the Microsoft .NET technology, we were able to provide a high degree of stability and usability and utilizing C# and third party libraries helped to increase productivity. We have made great effort to provide an architecturally sound design, resulting in a highly modular and maintainable system structure. Using Microsoft's LINQ technology, we provided a highly declarative way of defining metrics so as to accommodate the expected need of more and different metrics in the future. We have researched and picked out libraries (section 5.2.1) that will ease the process of implementing new visualisations in the future while keeping consistency in both implementation and visual style. We have also provided a manual that focuses on this aspect (appendix A).

7.2 Discussion

The project started out with two main application scenarios defined: on the one hand, ModelScope should be a tool for directly exploring models as in for instance allowing students to explore models in courses where models plays a major part (e.g. the course taught by the thesis supervisor). This creates high demands in terms of stability and usability. We believe we have met these demands especially as a consequence of the underlying technology and sound design principles. We have briefly explored this application scenario ourselves and with the help of fellow students, and preliminary evidence suggests we have indeed met this objective. A decisive proof, however, is not provided and can only be achieved in field test, e.g. in the upcoming course "Model Based Software Development".

On the other hand, ModelScope should be a tool for working scientists interested in studying models. This creates demands on the available probes, readings and visualisations and their extensibility. While a number of these have been implemented already, we are convinced that the set provided is far from complete and sufficient. Exactly which ones are the right ones is hard to determine up front, and only practical usage over a sustained period of time will reveal this. We excitedly expect what novel usage scenarios and measurement requirements will arise as the tool is used "in the field": will these fall outside the scenarios and test cases we have defined? It is this consideration that led to the implementation of "Exploration Mode" - a perspective for defining metrics and conducting measurements without visualisations. We believe this mode will prove useful in coming up with new probes visualisations and serve as an important link in this iterative creative process.

7.3 Future Work

Clearly, ModelScope needs a field test to detect shortcomings and weaknesses. We believe that the usefulness of ModelScope as a research tool will primarily rely on the repertoire of metrics and visualisations, and extending them permanently will be a perpetual requirement. Also, providing a richer and more tactile user interface might support the research flow. The modular architecture of ModelScope should welcome such future efforts in improving usability, even by for example using tablets or other platforms for the front-end.

The exploration of qualitative metrics could also benefit from a more dynamic data structure, which could allow for extra contextual data to be measured along with regular model data. This could be retrospective data like the number of people working on the model, hours spent, costs etc. Extending data structure and user interface with these options should be a reasonably manageable task.

Appendix A

Manual

A.1 Prerequisites

ModelScope runs on Microsoft Windows with .NET Framework 4.5. It should not be necessary to install framework manually if ModelScope are installed through the provided installer. If it should fail, .NET 4.5 can be downloaded from

http://www.microsoft.com/en-us/download/details.aspx?id=30653

A.2 Using ModelScope

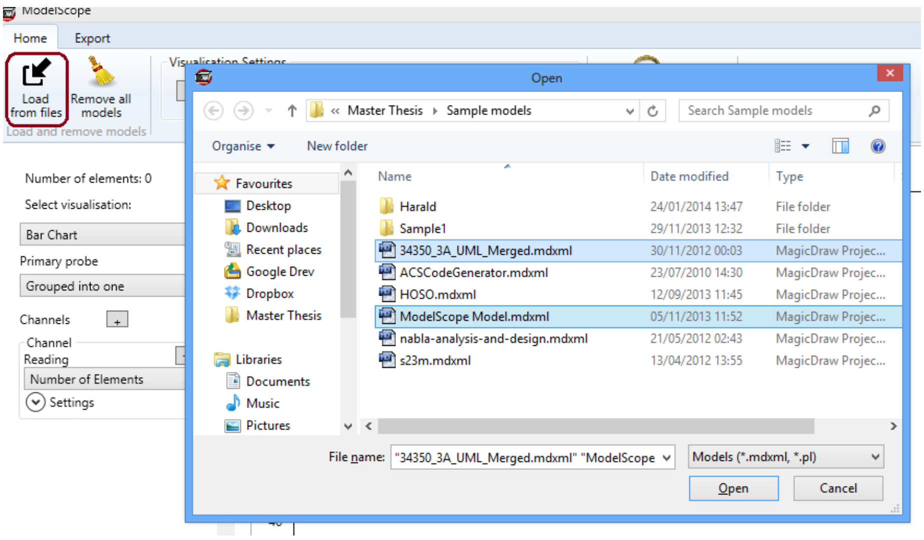
Most features of ModelScope should be fairly self-explanatory once the user has got accustomed to the terminology presented in this thesis. This section will describe how to do some of the most basic tasks in ModelScope.

A.2.1 Importing models

The current version of ModelScope handles two file formats:

- MagicDraw MDXML
- Prolog PL files from tools such as RED and Gourmand

Importing can be done at once ("bulk") or models can be added ad-hoc while working with the tool. Figure A.1 shows the dialogue to import models. Note that it is possible to add more models to ModelScope at any time in the work flow.



Number of elements: 0

Select visualisation: Bar Chart

Primary probe: Grouped into one

Channels

Channel Reading: Number of Elements

Settings

Name	Date modified	Type
Harald	24/01/2014 13:47	File folder
Sample1	29/11/2013 12:32	File folder
34350_3A_UML_Merged.mdxml	30/11/2012 00:03	MagicDraw Projec...
ACSCodeGenerator.mdxml	23/07/2010 14:30	MagicDraw Projec...
HOSO.mdxml	12/09/2013 11:45	MagicDraw Projec...
ModelScope Model.mdxml	05/11/2013 11:52	MagicDraw Projec...
nabla-analysis-and-design.mdxml	21/05/2012 02:43	MagicDraw Projec...
s23m.mdxml	13/04/2012 13:55	MagicDraw Projec...

File name: "34350_3A_UML_Merged.mdxml" "ModelScope

Models (*.mdxml, *.pl)

Figure A.1: Import models from .MDXML or .PL, one file at a time or multiple files at a time.

A.2.1.1 MagicDraw file formats

MagicDraw will typically save models as ".MDZIP" and not ".MDXML". ModelScope only reads ".MDXML" so it may be necessary to unzip the files before importing. This can be done via a zip-tool such as the freeware 7-zip.[1]

A.2.2 Using Visualisations with Probes and Readings

A.2.2.1 Zooming and Scaling

To zoom in or out an a visualisation, simply hold down the CTRL-key on the keyboard and use the scroll-wheel on the mouse.

To scale a certain part of the visualisation hold down CTRL and drag the mouse over the desired are while holding down the right key on the mouse. See Figure 6.2 and 6.3 on page 55 for an example of this.

A.2.3 Exporting

ModelScope has two ways of exporting:

- Visualisations as PDF
- Data of visualisations as CSV

Select the "Export" tab of the ribbon and click on either "Export diagram as PDF" or "Export diagram data as CSV" (Figure A.2). Note that it is the visualised data that will be exported to CSV, not the results of "Model Exploration Mode".

[1] 7-zip can be downloaded from http://www.7-zip.org/download.html

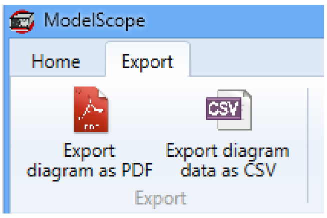


Figure A.2: The "Export" tab

A.3 Extending ModelScope

A.3.1 Set-up IDE and Environment

ModelScope is developed in Microsoft Visual Studio 2012 with .NET 4.5 as framework and Windows Presentation Foundation (WPF) as graphical interface. It has not been tested in other environments such as Mono on Linux, though it is possible that a port could be done.

To build and extend ModelScope locally, Visual Studio 2012 (or newer) needs to be installed. When installed, the ModelScope solution is opened through “ModelScope.sln”. Figure A.3 shows Visual Studio 2012 with ModelScope. To build the solution, click on “Start” next to “Debug” as shown in Figure A.3.

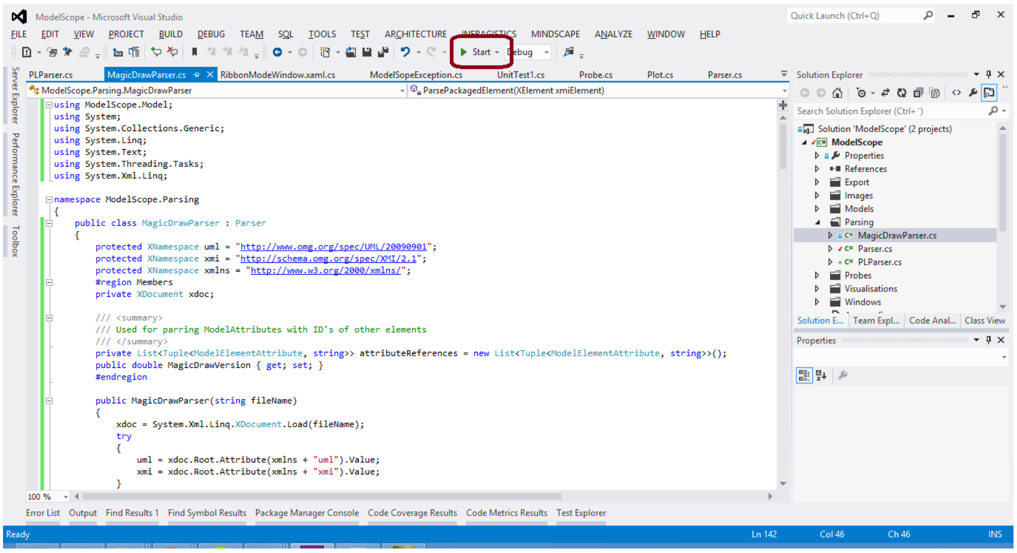
A.3.2 Making a New Probe

To make a new probe, right click on the folder “Probes” and select “Add”, then “Class...” (Figure A.4). Name the probe appropriately.

Make the new probe an implementation of the interface `Probe` by adding `: Probe` to the class as seen in Figure A.5. Then click the little little arrow on “Probe” and select “Implement abstract class Probe” to have Visual Studio automatically stub the class.

Name the probe by overriding the string “Name”:

```
public override string Name
{
   get { return "My new probe"; }
}
```



```
using ModelScope.Model;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Xml.Linq;

namespace ModelScope.Parsing
{
    public class MagicDrawParser : Parser
    {
        protected XNamespace uml = "http://www.omg.org/spec/UML/20090901";
        protected XNamespace xmi = "http://schema.omg.org/spec/XMI/2.1";
        protected XNamespace xmlns = "http://www.w3.org/2000/xmlns/";
        #region Members
        private XDocument xdoc;

        /// <summary>
        /// Used for parring ModelAttributes with ID's of other elements
        /// </summary>
        private List<Tuple<ModelElementAttribute, string>> attributeReferences = new List<Tuple<ModelElementAttribute, string>>();
        public double MagicDrawVersion { get; set; }
        #endregion

        public MagicDrawParser(string fileName)
        {
            xdoc = System.Xml.Linq.XDocument.Load(fileName);
            try
            {
                uml = xdoc.Root.Attribute(xmlns + "uml").Value;
                xmi = xdoc.Root.Attribute(xmlns + "xmi").Value;
            }
```

ModelScope - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ARCHITECTURE INFRAGISTICS MINDSCAPE ANALYZE WINDOW HELP
Start
Debug
Quick Launch (Ctrl+Q)
PLParser.cs
MagicDrawParser.cs
RibbonModeWindow.xaml.cs
ModelSopeException.cs
UnitTest1.cs
Probe.cs
Plot.cs
Parser.cs
ModelScope.Parsing.MagicDrawParser
ParsePackagedElement(XElement xmiElement)
Solution Explorer
Search Solution Explorer (Ctrl+")
Solution 'ModelScope' (2 projects)
ModelScope
Properties
References
Export
Images
Models
Parsing
MagicDrawParser.cs
Parser.cs
PLParser.cs
Probes
Visualisations
Windows
Solution E... Team Expl... Code Anal... Class View
Properties
Server Explorer
Performance Explorer
Toolbox
100 %
Error List Output Find Results 1 Find Symbol Results Package Manager Console Code Coverage Results Code Metrics Results Test Explorer
Ready
Ln 142 Col 46 Ch 46 INS

Figure A.3: Visual Studio 2012 with ModelScope code base

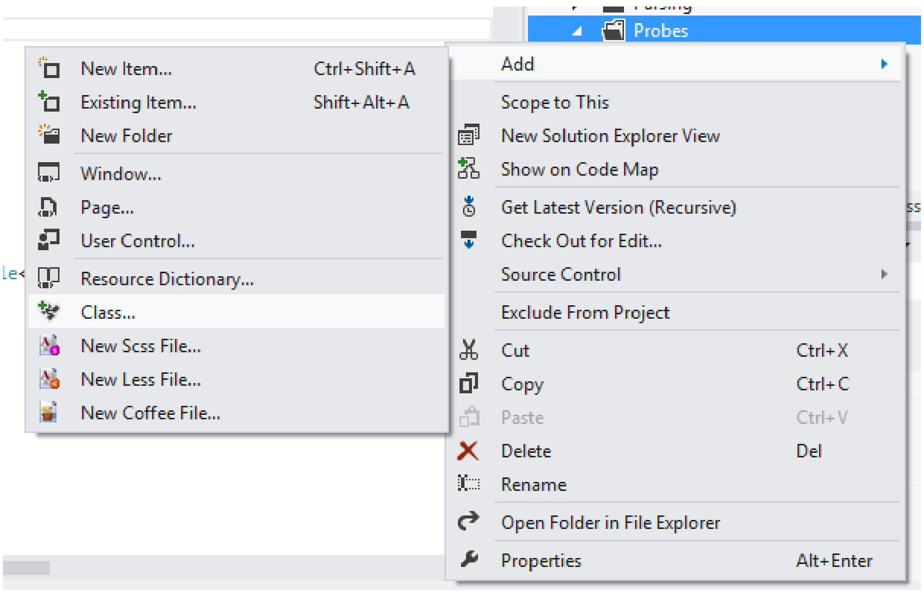
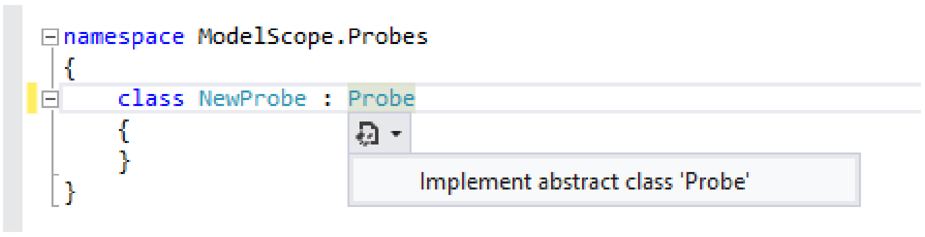


Figure A.4: Add a new probe



```
namespace ModelScope.Probes
{
    class NewProbe : Probe
    {
    }
}
```

Figure A.5: Visual Studio can automatically stub interfaces.

`ApplyProbe` has to return a `List` of `ProbeResults`:

```
public override List<ProbeResult> ApplyProbe(ProbeResult
    probeResult)
{
   var result = from element in probeResult.Elements
                    group element by element.Attributes.Count()
                        into numberOfAttributes
                    select new ProbeResult { Key =
                        numberOfAttributes.Key.ToString(),
                        Elements = numberOfAttributes.ToList() };

   return result.ToList();
}
```

This can be done using the declarative LINQ as in the example above or imperatively if desired.

Lastly, the probe needs to be added to the global list of probes in ModelScope. This is done in the static method `Probes.InitProbes()` by adding the following code:

```
AddProbe(new NewProbe());
```

Breakpoints can be added anywhere using F9. Start debugging by clicking "Start" as shown in Figure A.3.

A.3.3 Making a New Reading

To make a new reading, follow the instructions of how to make a new probe in section A.3.2. The process is essentially the same. Instead of implementing `Probe`, the new class should implement `Reading`.

Bibliography

[CL07] Michel R. V. Chaudron and Christian F. J. Lange. Second international workshop on model size metrics. In *MoDELS Workshops*, pages 89–92, 2007.

[FP98] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.

[Sto10] Harald Storrle. Towards clone detection in uml domain models. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 285–293, New York, NY, USA, 2010. ACM.