ROBOTICS AND MECHATRONICS

Improving the Model Management Workflow

T. (Tom) Bokhove

MSc Report

July 2016

015RAM2016
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

controllab

UNIVERSITY OF TWENTE.

MIRA BIOMEDICAL TECHNOLOGY AND TECHNICAL MEDICINE

CTIT

Table of contents

Acknowledgements

During the scope of my master assignment, I got quite some help. First of all I would like to thank everyone at Controllab Products B.V. for their help, ongoing support and trust. In particular, I would like to thank Despina Davoudani as daily supervisor for the first part of my master assignment, and her help in building the initial version of the Graphical User Interface. I also would like to thank Maarten Meijer for continuing the job of daily supervisor for the second half of the master assignment, and for his help during the XML and Smalltalk aspects of my master assignment. Another thanks goes to Marcel Groothuis for his help on the software aspect of my master assignment, his advise on several software issues like the choice for a proper version control system and his council for working with Visual Studio helped a lot. Furthermore, I would like to thank Paul Weustink for his help during the impact analysis for obtaining the set of generated equations, and for his feedback. Finally, I would like to thank Christian Kleijn and Frank Groen for their ongoing feedback about respectively the user-aspects of the tool and C++ knowledge.

I would also like to thank the University of Twente for making my master assignment possible, especially the chair of Robotics and Mechatronics, which made it possible for me to do my master assignment at Controllab Products B.V. In particular I would like to thank dr. ir. P.C. Breedveld for the progressive discussions during the monthly feedback sessions and his role as university supervisor. Furthermore I would like to thank ir. E. Molenkamp for his guiding role as supervisor from the Embedded Systems chair.

Abstract

Among the most recent developments in software engineering is the tendency to use a model-driven approach to give a visual representation of the system under development. One will however quickly notice that working with models takes quite some management to be effective. Every minor change might become a new version of that model, and every important version might have to be uniquely labelled to account for this. The challenges become even more complex if multiple persons are working together on the same model, since different versions of the same model exist in parallel, that eventually have to be merged again into one model.

During the research phase, it became quickly clear that version control was the way to go for managing these models. The choice was made to use an existing version control system, namely GIT. The focus for model management was placed on the tool 20-sim, a modelling tool for physical systems. 20-sim uses one file that contains all the information that the model needs to be loaded and simulated. Most of the version control actions also directly apply to this model file, like storing a commit (a version of a model) and making an alternate copy of a model (a branch), but the problem lies in merging two models, since the resulting model should be a valid model as well.

The practical phase was mainly focused on doing exactly that: analysing and categorising differences between model versions and merging these model versions together. Not just providing a textual difference between two models, but really going in-depth in the context of the different aspects of this 20-sim model by using a metamodel specification of what a model should contain to be valid. By interpreting the actual components in a 20-sim model, like terms as submodel, plot, and connection, it becomes directly possible to compare these contextual components over several versions of the same model.

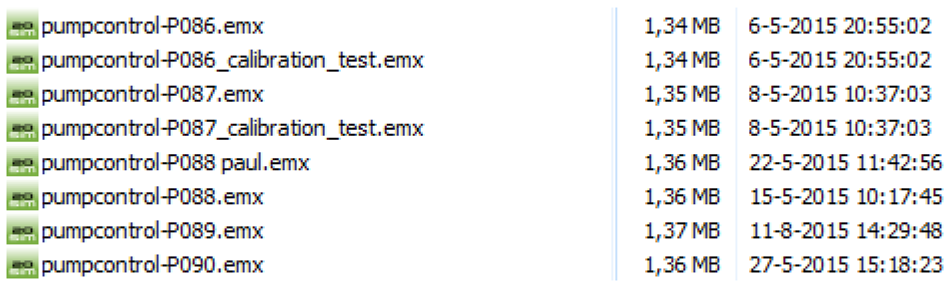
The unique feature of this tool, in which it goes further than any of its competitors on the market, is that it also provides a mechanism for determining the impact of a difference between two model versions, even before the merge is performed. This impact is determined based on two methods: Finding out if a set of equations has operationally changed, and trying to find an impact score for a certain difference based on what the user is likely considering a high impact difference.

The comparison tool itself is built as a proof of concept for this master assignment. The tool works, and is able to determine the differences and perform a merge based on a large variety of models. The tool is linear in both the time it needs to process the differences, and the memory it allocates for this process when looking at the size or amount of equations of the models initially fed into the tool. The most important recommendation is to monitor model transformations in 20-sim. These model transformations are the atomic actions that a user takes to alter a model. By monitoring these actions, it becomes possible to deduce the differences directly from the model once the user adds them to that model. Furthermore, determining the impact of a difference on model behaviour was found to be a promising field of research in the sense that it can give the user an indication on what differences made between model versions are the most important.

Chapter 1

Introduction

Collaboration can be beneficial to the results of a project, but it poses challenges too. One of the fields in which little research has been done in managing collaboration is the field of computer-aided, model-driven engineering. The target of this field is to develop models as the main source of engineering with the goal to simplify the development process. Model management is a difficult issue, especially when many users work on the same model. An example of an existing project, in which a pump controller was designed, is shown in Figure 1.1. This is just part of a very long list of model versions with all kinds of version indications. The most direct version indication is the number of the model, in the figure ranging from 086 to 090. However, some models have multiple versions that are distinguished by an additional note, rather than by creating a new version. Other models indicate the version of a specific person, to indicate the author of the newly created version. Note that there was even a modification in the model with version number 089 that was modified after the last change to version 090, which can be observed by looking at the timestamps of both versions.



pumpcontrol-P086.emx	1,34 MB	6-5-2015 20:55:02
pumpcontrol-P086_calibration_test.emx	1,34 MB	6-5-2015 20:55:02
pumpcontrol-P087.emx	1,35 MB	8-5-2015 10:37:03
pumpcontrol-P087_calibration_test.emx	1,35 MB	8-5-2015 10:37:03
pumpcontrol-P088 paul.emx	1,36 MB	22-5-2015 11:42:56
pumpcontrol-P088.emx	1,36 MB	15-5-2015 10:17:45
pumpcontrol-P089.emx	1,37 MB	11-8-2015 14:29:48
pumpcontrol-P090.emx	1,36 MB	27-5-2015 15:18:23

Figure 1.1: An example of a project in which collaboration on the same model resulted in quite some versions of a model.

The question becomes how to interpret these results from an existing project. Apparently some versions were labelled with additional information, which indicates that it is a special version of the model (”calibration_test” is such an example). Sometimes there are also alternate timelines for the models, in which the project members take the same model as starting point, but both adapt it into their own version. A name then indicates whose version it is. These models are usually likely candidates to get merged at some point with the main line of models, since the project should eventually deliver a final model as product to the customer. Given all these indications on model versions, it becomes cluttered what happens in one specific version. What is for example the difference between version 089 and 090? What is the actual meaning of the label ”calibration_test” to model version 086, since version 086 also exists without the label? These labels and numbers might thus be differently interpreted, and give no information about the contents of the version at hand. To summarise, the question thus is if there is a way to improve this model management workflow and increase insight in the different versions of a model.

To capture the problem statement in a more formal way, a set of requirements about the model management workflow is set up, as shown below:

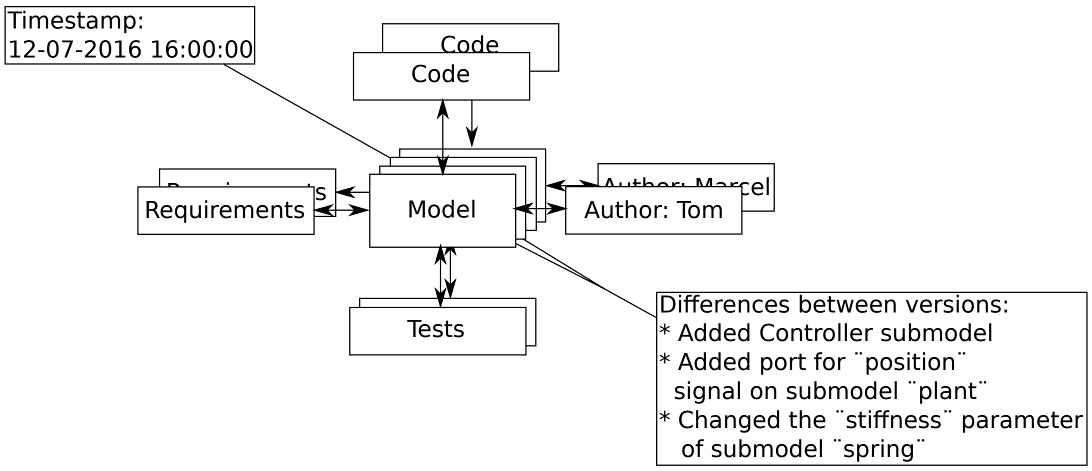

Timestamp:
12-07-2016 16:00:00
Code
Code
Requirements
Requirements
Model
Author: Marcel
Author: Tom
Tests
Differences between versions:
* Added Controller submodel
* Added port for ¨position¨ signal on submodel ¨plant¨
* Changed the ¨stiffness¨ parameter of submodel ¨spring¨

Figure 1.2: The context of a model in a project.

Requirements 1.0.1: Model Management Workflow

- There should be a means of storing the versions of a model automatically.
- It should be possible to label a specific version of a model as important.
- It should be possible to see the differences between versions.
- Versions should contain information about the time that they are created, and the person that made the version.
- It should be possible to merge versions of models that grew apart due to several users adapting the same model separate from each other.

1.1 Goal

The goal and its subgoals for this master assignment can be described as follows:

- Goal: To ease the proces of model management for a user that would like to focus on the development of models rather than the management of them.
  - Subgoal 1: give the user a means of easily storing and retrieving versions of a model
  - Subgoal 2: give the user a means to compare two versions of the same model
  - Subgoal 3: give the user a means to link his model to external factors as results, code, tests, and requirements

The first subgoal implies that the user should easily be able to indicate that something is a new version, store that new version, and later on be able to retrieve that same version. However, model management also includes some transparency. Not only does the user want a list of versions, there is also a need to see what actually happened in between these two versions. That is where subgoal 2 comes into play. The final subgoal relates to the fact that it is quite common that a model is part of a larger project. An example of such a project is shown in Figure 1.2. The first thing to note is that the project is expanded by things like requirements, tests, and generated code. The second thing is that the context of the model is also expanded by the changes made, the author of these changes, and the timestamp at which time these changes were stored.

1.2 Justification

Even though modelling is nowadays a common way of development, model management has not reached much attention yet. This master assignment will start by evaluating the available literature on model management. The justification of this master assignment lies in using this to build an actual model management system around a commercial tool. As will become clear in the literature research chapter (section 2.5), there are not many commercial tools that apply model management or dedicated comparison techniques of models yet in their tool. The tool of 20-sim (2016) will be used as a starting point to try to use several scientific papers and their theories on model management in practice. Furthermore, a level of impact between model versions will be estimated that indicates how large the impact is on the simulation results of the model based on the differences between these two models. This impact analysis is something that none of the commercial tool vendors as of yet have implemented in their tool.

1.3 Overview of the report

The literature research (chapter 2) will focus on finding literature about model management. This literature research will be extended by doing research into placing the model in its project context, as was shown in Figure 1.2. The design chapter (chapter 3) will closely relate to this, by using knowledge from the literature research chapter to make a design for model management in 20-sim. The implementation chapter (chapter 4) will mainly focus on the implementation of a model comparison tool proof of concept, that lies at the base of model management. This tool will also be the main point of evaluation in the results chapter (chapter 5) that follows. In this chapter, the focus will be placed on trying out this model comparison tool for a large test set of models. Finally, the report will be concluded with a discussion (chapter 6) and conclusion (chapter 7).

Chapter 2

Literature Research

The introduction already shortly touched upon the problem statement of this report. The challenge lies in the management of model versions. The first useful topic is to look into version control for model-driven projects. Is there any research done on model version control, and can it give leads to finding solutions to the problem statement? The next step is to look in a concept called traceability. A model is often much more than just the model itself. Connections to requirements, generated code samples or resulting simulation plots usually are specific to a version of a model. If the model changes, these connected aspects should possibly change their behaviour too to match the new model version. Afterwards the concept of provenance is investigated, in which the topic of gathering data about a model version is the central theme. What information is crucial for identifying a specific version of a model in a large set of versions of that model? Provenance is followed by a section about impact of differences between model versions on the overall simulation results of that model. Finally, existing commercial tools will be evaluated.

2.1 Model version control

Version control is a concept that gets more and more attention lately. The development of newer and more robust Version Control Systems (VCS) makes the options for version control more applicable to different kinds of software applications too. Version control in essence is concerned with storing a version of a file, piece of code or project for later reference. The demand for model-driven version control is however also increasing, even though the development of this branch of version control seems to lack behind. Models are not just blocks of sequential code, they contain functional units interconnected with a set of connections. The main challenge here lies in identifying components of a model that can undergo an atomic operation. The unique identification of elements, the atomic model transformations, and conflict resolving are among the main problems with model based version control. These topics will each be discussed in this section.

Brosch et al. (2010) describe a framework that was developed for model-driven version control. Their model versioning system, AMOR, is a structure of steps on how to find conflicts, and resolve those conflicts in an automated manner. The conflict detection happens by comparing the new versions of the model against the original model and against each other. A set of operations is defined, which among others contains "insert", "update" and "delete" model elements. Additional operations can be defined by the user. After detecting conflicts, the system tries to find a fitting resolution strategy which can automatically resolve the conflict. If there is no such strategy, the user is asked to manually resolve the conflict. Manual resolutions are then evaluated by the system, to see if any automatic resolution strategies can be obtained from this for future conflicts. Finally, the actual merging phase is done, after which only one version will remain.

Reiter et al. (2007) describe a challenge in model-driven engineering related to the semantics of models. Changing a minor thing in a model, can have major impact on the behaviour of the model. They designed a way of versioning models that should be able to avoid this problem, by not only doing a syntactic analysis (evaluate the differences between two models), but also by doing a semantic analysis (evaluating the impact of these changes on model behaviour).

Wenzel (2014) describes a problem that is shown throughout quite some scientific papers about model-driven versioning, which is the problem of unique identification of elements that should be under versioning. To define an element in a unique manner, one has to assign a unique identifier to every component in the model that might possibly change behaviour or implementation over time. However, if such a component is given a unique identifier, then it keeps that identifier until the end. It is then often likely that this component is completely different from its initial implementation, even though it is referred to with the same identifier. Wenzel suggests to use traceable links that link attributes, references, movements, and structurally different components, such that not the component itself, but rather its context (attributes, links, and implementation) are monitored.

Finally, one of the hints that came forward from many papers that describe model-driven revisioning is that so-called model-transformations should be monitored, rather than only the changes between revisions. In that way, one could better evaluate the difference in behaviour between models. By evaluating a step-by-step progress report of a model, one can see in-depth which components were added or removed, and which ones were changed. An example of such an article that states the importance of model transformations in model-driven version control is created by (Taentzer et al. (2014)).

To summarise, the challenges with model-driven version control lie in resolution strategies to automatically solve conflicts, including semantics of models in versioning, identifying components in a unique and complete manner, and using model transformations to identify actual changes to the model in a more atomic manner. Some of these challenges will be discussed in the Design chapter, as will be introduced in section 2.6.

2.2 Traceability

According to IEEE std 610.12-1990 (2002), traceability is defined as: "The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match." This definition indicates the major aspect of traceability: combining software artefacts via a set of traces, such that it is possible to see what elements of a project are connected. The example of a requirement to part of a design is such an example that might be relevant for model-driven traceability. This can however be further extended to a set of tests to verify a requirement or a piece of documentation that is coupled to part of the model.

Winkler and von Pilgrim (2010) made a survey about traceability in model-driven engineering, in which four phases were derived from literature: Planning and preparing, Recording, Using traces, and Maintaining. The planning and preparing phase is concerned with making a traceability scheme that identifies the elements and their relationships. The recording phase implements the traceability scheme onto the actual elements and links between those in the models. Using the traces means that data can be taken from those traces that might be used for generating documentation, finding relevant information about the connection between models and their elements, and maintaining model consistency. Once everything is in place, it should also be maintained. If elements are added, deleted, changed or duplicated, this means that their corresponding traces also change. This is the primary goal of the maintenance phase.

Schwarz et al. (2010) suggest additional phases like identification (using automatically found relationships to derive additional relationships) and retrieval (getting the trace information back), however these can be placed within the four phases defined above. Schwarz a.o. did however make an additional distinction in intra-level traceability (traceability at one level of a software architecture) versus inter-level traceability (traceability that ranges over multiple levels of a software architecture).

Hull et al. (2005) describe in their book a chapter about traceability, in which they also define some benchmarks to traceability over multiple levels of a software architecture. The first is breadth: to what extent are the requirements in the architectural level above covered by the current layer of requirements? The second is depth: how deep are the requirements propagating in the model (i.e. in what sublevel of a software architecture is the requirement needed)? The final one is growth: what is the impact of a change of a requirement to the other requirements throughout the system?

2.3 Provenance

According to Cheney et al. (2008), provenance can be described as: "information describing the origin, derivation, history, custody, or context of an object". The word provenance originally was used for tracing paintings back in time over history to their creator. This application of provenance also describes rather well the purpose of it in software engineering: tracing versions of models over time. In that sense, it also has quite some overlap with model-driven revision control and traceability. Its main distinction as a concept from revision control an traceability is that it is focused on what to store as metadata corresponding to software artefacts, rather than the storage of the software artefact itself. Provenance is also very applicable to several domains, like scientific workflows, databases, security, and bioinformatics.

Cheney et al. (2008) provide some challenges that were discussed during their provenance workshop. Their first concern is that the direct benefit of provenance is not directly clear. This also will make it unlikely that users maintain the provenance tags to their software artefacts on the long term. Another challenge is to determine what to track and who to allow to use these tracking tags for data. This latter option of who is allowed to watch the provenance data, is often used in security applications that make use of provenance. In model-driven design and architectures specifically, the maintenance of model transformations, as well as the propagation of changes in models were once more mentioned as important aspects, also for provenance (these were introduced in section 2.1).

Provenance is described by Shamdasani et al. (2014) in three aspects: tracking changes to provide historical records of actions performed, the outcomes achieved, and design decisions taken. They use an XML scheme to couple this additional data to a model. The main usage for their application of CRISTAL is event-driven and state-based. Their main focus is on workflows (which are directed graphs, with states and events). There is a lot of provenance data stored for each state for each revision: event ID, activity name, previous and target state, transition, outcome scheme name and version, agent name, agent role, and time stamp. An agent is defined as a human user or automated system that operates on the model.

Another approach towards provenance is a more formal one by Dezani-Ciancaglini et al. (2012). Their approach is to formalise the mathematics behind provenance in a triple based on linked data management. This triple consists out of the following elements: subject, property, and object. The subject is the place or repository in which to look for data, the property is the type of search that is done, and the object is the search term. For example: Look for data in the subject "University library" on property "year of publication" and object "2012".

Besides their formal algebra, Dezani-Ciancaglini et al. (2012) also define types of provenance: "where now provenance" (next to their triple of data, a location is added of where the data is currently located), "who now provenance" (next to their triple of data, an agent that published this data is added), and "how provenance" (recording the operations applied to the triple of data).

2.4 Determining an impact

In model management, the impact of the differences between two model versions is an important indicator in seeing how relevant a certain change is. Therefore, as an additional feature to the model comparison tool, some investigation was done in two types of impact. The first one was already mentioned by Reiter et al. (2007), and is detecting not only that a difference occurs in the equations of a submodel, but also keeping into accounts the semantics of the model. In that way, it becomes possible to see if there is an actual operational difference, that might also affect other parts of the model that are connected to that submodel. The design chapter will couple this part of the impact to the rest of the master assignment (section 2.6).

Furthermore, a technique called Bayesian learning will be used to estimate a percentage of impact. Bayesian learning is a technique taken from the domain of artificial intelligence, and it computes the probability that a certain impact score is what the user wants to see, based on a set of learning examples that include information about the component of the model that changed. This set of learning examples can be expanded by learning examples that the user chooses to include. This means that the impact scores that are given, will be biased by the wishes of the user. More information about this Bayesian Learning technique is shown in Appendix D, which is based on the articles of Yang (1997), Do and Batzoglou (2008) and Osoba et al. (2011).

2.5 Commercial model comparison tools

There are quite some theoretical tool ideas available for model comparison according to modeling-languages.com (2010), but there are only few actual commercial applications that use model-driven version control. This master assignment was not done to add another theoretical tool to this list, but to see if it is possible to use this research and apply it to an actual commercial tool.

In line with the rest of this report, commercial tools that properly compare two models are limited. To specify what models are the main point of investigation, the choice was made to mainly look at competitors of 20-sim, thus physical modelling tools. The main limitation on most available tools is that they do not provide the model context to the user, but simply perform an XML-comparison. They do not show the submodels, ports or connections that changed, but rather a set of XML tags that changed between two model versions. There are a few examples of such commercial model merging tools, however they are sporadic and mostly focussed on the program Simulink. Mathworks does have their own tool that can show the XML differences between the .mdl files that are created in Simulink, but also does not provide context. A more commercial tool is SimDiff (made by EnSoft Corporation (2005)), another Simulink model comparison tool. This tool however does provide a layer of context by indicating the submodels in which a difference occurred, and by providing a mechanism for merging simulink submodels. It is possible to either merge two models into a new Simulink model or merge another model into the currently opened model. Furthermore, SimDiff has the possibility of showing differences at a graphical level by a colour indication of the changed submodels.

2.6 Relation to design chapter

The literature gave some interesting leads for the rest of this assignment. One of the main topics in model-driven version control was to include semantics in merged models. Does the simulation result of the model change due to the differences found between both versions of the model? This directly couples to traceability. For example, if a submodel changes, then the set of traces to for example corresponding unit tests can become outdated. One step further down the road, it might even be possible to regenerate these results based on invalidating outdated traces to this specific submodel. Shamdasani et al. (2014) go on with provenance that it is not only historical tracking of data, but also monitoring the outcomes. Combining these three topics, one could see one main line of reasoning: A model is not only the model file, but it is the full combination of the model, its requirements, tests, documentation and its results. In the ideal case, one would thus not only merge the model, but also regenerate the results of that model if the semantics of the model change. An initial proof of concept of this phenomenon is described in the Impact part of this chapter in section 3.5.

Another part is the impact aspect. A technique from the artificial intelligence field is used to make an estimation of an impact score of a changed submodel. This estimation will be based on several properties of a submodel, like the fact that a port was added or deleted or the fact that the submodel operationally changed its behaviour. This part will be treated in the Impact part of this chapter in section 3.5.

The final point from the literature research concerns model-driven development. An idea mentioned was to use model transformations as a way of identifying atomic actions, such that the exact differences could be seen between two different versions of a model. Even though this idea is not literally used in this report, the concept itself of identifying atomic actions was used to come up with an idea on how to provide context to an otherwise meaningless XML-file representing a model. The idea of identifying blocks that have a certain meaning, and that can undergo certain transformations was used to identify the basic outline for what parts of a model could change. In that way it becomes possible to only show differences that are interesting for the user to see in their list of differences between two model versions. The basic set of operations, namely insertion, deletion and modification, was already mentioned by Brosch et al. (2010), and can be reused for these contextual blocks. The relation to the application developed in this report will be explained in section 3.4.

Chapter 3

Design

Model management is quite a broad concept. This chapter will introduce which parts of model management are going to be covered in the rest of this report. The main idea is to develop the tool support needed to use a regular version control system for model-driven version control. As an extension to regular version control systems, an additional impact analysis will be performed, that evaluates the differences and tries to find the impact on the simulation results of the model. An overview flowchart is shown in Figure 3.1.

This overview diagram will be the basics of the rest of this chapter. section 3.1 will go into more details about the use-case scenarios mentioned in the Model Management block of the figure. The version control section is described in section 3.2. In this section an evaluation is done on the most promising existing version control system to be used in model version control. Furthermore, this section explains a technique called subtree merging, needed for proper model-based revision control at model level. The block "Version Control System" in the figure shows a basic flowchart of the version control, in which all use-cases first relate to the version storage database, and afterwards the decision is taken to call the Tool block or not. The Tool block itself in the figure represents the Tool designed during this master assignment. The blocks "Obtain XML tree", "Add Model Context", "Find Differences", and "Merge Models" are described in section 3.4. The block "Determine Impact" is further elaborated on in section 3.5. For the implementation chapter (chapter 4), the tool 20-sim will be used as modelling tool for implementing this flowchart. For more information about the tool 20-sim, see Appendix C.

Before continuing with the rest of this chapter, the text above will be formalised in a set of requirements, that are the core reason to write the rest of this chapter. These requirements are added to the already presented list of requirements given in chapter 1 in requirements block chapter 1.1.

Requirements 3.0.1: Design

- Where possible, the model-driven version control should make use of existing functionality already available in file-based version control systems.
- It should be possible to keep track of the history of every separate model, and also of the folder containing the models.
- Based on increasing the effectiveness of model management, the graphical user interface should include ways of giving users feedback on the impact that differences between model versions might have.
- The implementation will be done in 20-sim, which means that a dialogue has to be made that handles the basic interaction with 20-sim in an intuitive manner.

3.1 Use-cases

During modelling, users can perform certain actions that are relevant to be monitored for a version control system. It is the challenge though to monitor these actions without bothering the user. Some of the actions to monitor are for example when a user saves a model (or actually creates a new model version) or opens another model. These actions together can give the version control system the indication of when

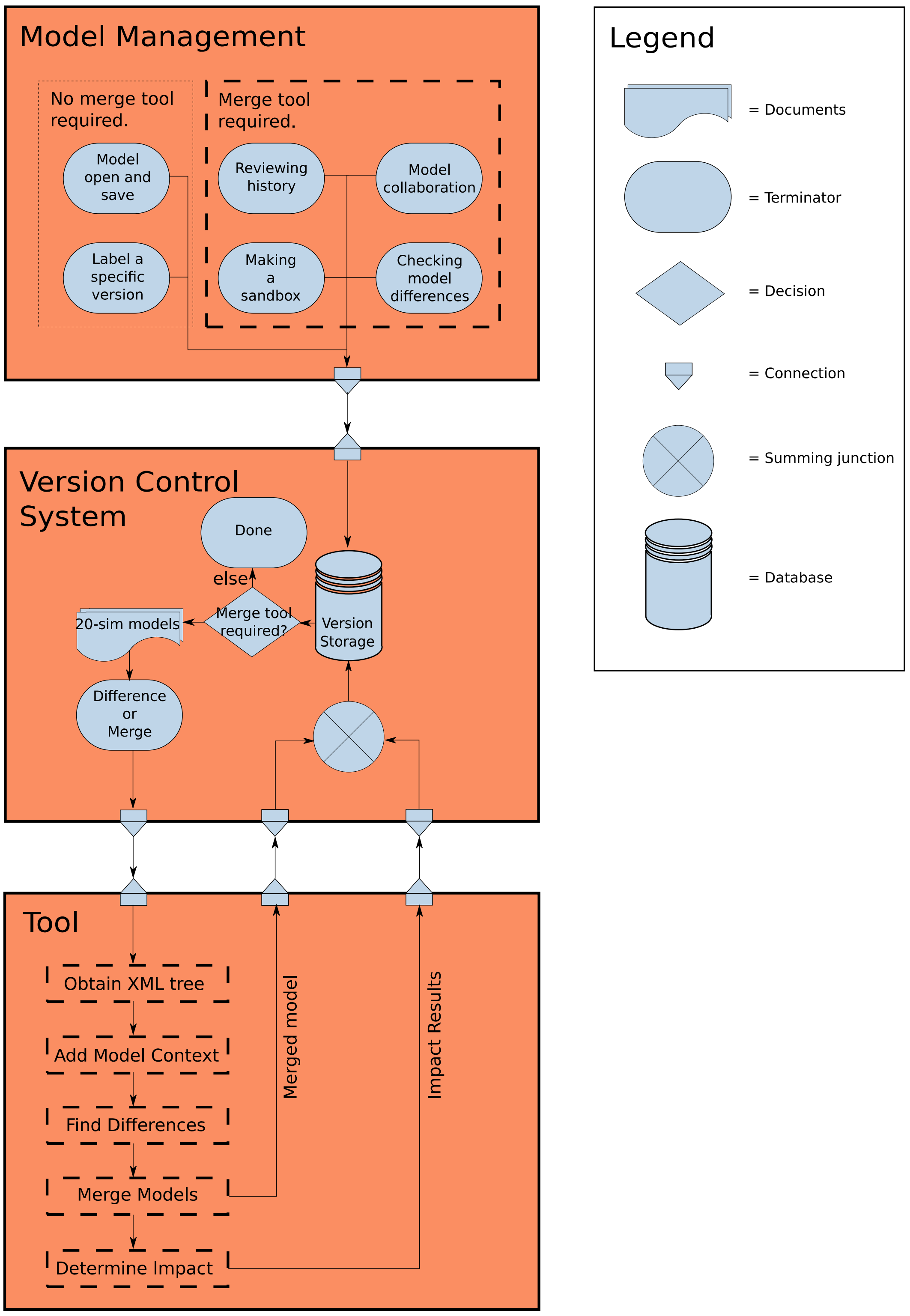


Figure 3.1: The flowchart of the Design phase of the project.

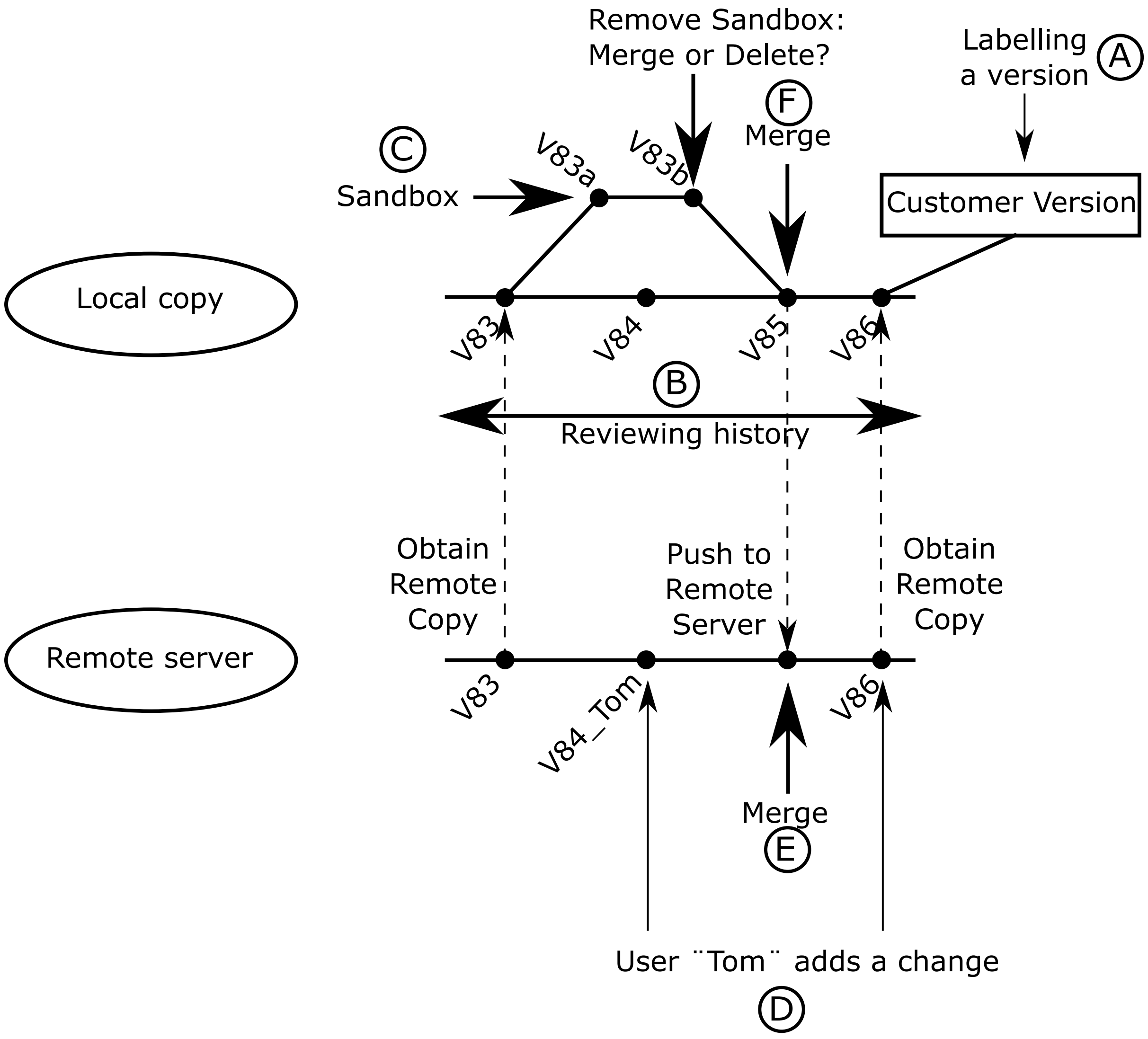


Figure 3.2: A graphical representation of the use-cases.

to create a new version, and where to obtain requested versions from. The results of this monitoring can be used in various cases, like reviewing the history of a model or merging models together.

Add/Open/Save/Save As The four basic actions for creating opening and storing models are accessible to users. The main idea behind this use-case, is that the user should be able to press these buttons without having the feeling that under the hood they are doing revision control. On the other hand, the user should get access to the features that are related to version control, like reviewing the history, which needs monitoring of when the user presses these buttons.

Label a special version It should be possible to label certain projects or models via a tag. For example, a tag could be "Final version", "V1.0", or "customer version". By using these labels, it should become easier to retrieve that exact model or project revision later on. A dialogue with a history line for all versions could for example contain a list with all these special labels, such that upon clicking on a specific label it will automatically go to that point in time. Figure 3.2 for example shows in the local copy that V86 is labelled with the tag "Customer Version", as indicated by the letter A.

Reviewing history The reason for applying revisioning to 20-sim, is to be able to view the history of a model. Via an intuitive dialogue, the user should be able to go back in time. For example, a slider could be used to see the model itself change graphically over time. By going through previous versions of a model, it is possible with revision control to make a copy of a model in a previous revision, and adapt that specific copy of the model. One could for example model a car, and review the previous versions of that modelled car. It is also possible to take a certain version in history of that car, and adapt it in a copy of that version of the model. Figure 3.2 shows a graphical depiction in which a timeline is shown along the horizontal axis, indicated by the letter B. Reviewing history would mean that one could go back and forth through time to visit older versions of a model and compare them with newer versions of that same model.

Making a sandbox of a model A sandbox is meant for quickly trying something out in a model. A simple example is a basic control loop. At some point, it might be useful to evaluate a different type of controller. The current approach is to either make a copy of the model and store it under a different name or add the controller as another implementation to the already existing controller. A sandbox makes a temporary copy of a model, in which it is possible to add the new controller. The simulation results can then be evaluated, and a decision can be made about deleting all changes in the sandbox or merging them with the original model. In that way there is no need to change the original model with possibly failing submodels, until they have proven themselves to be useful. Figure 3.2 shows one possibility of a sandbox, indicated by the letter C. The local version V83 splits up into a version V83a. Version V83a is an adaptation of V83, and V83b is an adaptation of version V83a. At some point, the sandbox version V83b is ready, and the user decides to merge back the version V83b with the then applicable version V85. A merge is needed to combine the new behaviour of the sandbox, with the main line of history of the local copy.

Model collaboration For user collaboration, it should be possible to identify a shared location as shared project. When a user has a network connection, these changes can then be placed in a queue on the shared location. Once the user sees his changes as useful to share with others, it is possible to merge all or a set of his changes with the overall shared project. In this way, easy collaboration can be achieved within projects that have more than one person working on the same model. Figure 3.2 shows several points of remote to local or local to remote connections. These are indicated by dashed arrows between the timeline of the local copy and the timeline of the remote server. Note that it is possible that another user, in this case user Tom (indicated by the letter D in the figure), alters the history on the server. This means that a merge of Tom his version and the user its version should be merged to obtain the next server version (indicated by the letter E). Another more local merge is shown in the same figure by the letter F.

Checking model differences Besides all kinds of version control related use-cases, it might also be convenient to use the tool for regular usage in 20-sim. It is often insightful to have two models, and just view the differences between these two models or merge these two models into one model. There should be a dialogue that shows this result in 20-sim in a proper and intuitive manner.

3.2 Version control

Version control lies at the basics of file management, but it is also a crucial part of model management. Instead of putting tens of versions in one folder (for example, see Figure 1.1), to which everyone can make changes at any time, it becomes possible to check out one model. This makes sure that you work on one version at a time, and the version control system will make sure that merge conflicts due to collaboration on the same model will be properly resolved (most likely with user interaction). This merge seems to be the main problem though, since model merging is different from file-based merging. Brosch et al. (2010) and Taentzer et al. (2014) were already mentioning the difficulties with comparison of models for the use of version control, in the sense that model transformations have to be monitored, and merge conflicts often have to resolved by relying on model-based merging algorithms. On the other hand, all the other functionality of the version control system is almost identical for both file-based and model-based versioning. "Committing" a new version of a file or a model is exactly the same. Opening a specific version of a model or file is also identical. Even remote sharing of models is the same as files, provided that a proper merging tool exists. Therefore it does not pay off to write a fully new version control system for model-driven version control, if most of the functionality is already available for models too. The section below will therefore evaluate existing version control systems, and see which one matches the needs of model-driven version control the best.

3.2.1 Version control system options

The three most common version control systems for file management are GIT, SubVersion (SVN) and Mercurial. These three version control systems will be compared to see which one will fit the needs for model-driven version control the best. The table with the results of a design space exploration are shown in Table 3.1. The references to the user manuals are also found in this table. These user manuals are used to base this design space exploration on. The scores each range from 1 till 5, with 1 being the worst

Criterion:	GIT [1]	Subversion [2]	Mercurial [3]
Hybrid	5	3	5
Conflict-avoidance	5	5	5
Development-status	5	5	5
Running under Microsoft Windows	5	5	5
Embedding in other tool	5	4	4
License conditions	3	5	1
Overall Score	4.7	4.5	4.2

Table 3.1: The Design Space Exploration for the Version Control software.

possible score, and 5 being the best possible score. The overall score is the average of the score in the rows above, and is the final score on which the version control system will be chosen.

First of all, the version control system should work under Windows, because it should cooperate with the tool 20-sim that only runs under Windows. This requirement is satisfied by all three version control systems. Furthermore, the version control programs should be able to be embedded within the application analysed in this master assignment. Also this embedding in another tool goes fine for all three version control systems. Git has a library version called libgit 2. This set of libraries is fully written in C++, and thus can be easily embedded with the C++ code for this application. C++ is not necessarily a requirement for the tool, however it is a language that supports decent object-oriented programming, which is useful for the tools to be developed in the Implementation chapter. Furthermore, C++ is a language for which the most experience is present in the company at which I do my master assignment. Subversion is already written in C, and thus can also be used in existing C++ applications. Mercurial is completely written in Python, which works easily together with C++ too. The only reason to give GIT a 5, and the other systems a 4, is that GIT directly works with C++, without having to do a conversion from another language.

License conditions are a criterion for commercial tools as well. GPL, the General Public License, is a license that states that any tool that includes this piece of code in their application should also make their own source code of their software publically available. Mercurial falls under this license, and thus gets a score of a 1. GIT falls under the GPL license too. However, libgit2 does not. Libgit 2 falls under the GPL license with an extension for linking the source code. Thus as long as the source code is not altered and just compiled along with the source code of the application developed, then the source code of this latter application can remain private. The Apache license, under which SubVersion falls, is the best license condition for commercial software, since it puts no restrictions on putting own source code online when linking against programs falling under this license.

The development status of the version control systems is important, since if there has been no recent development on the tool, it is likely that this tool might not get any bug fixes or new updates anymore. At the moment of writing (12 June 2016), the latest version of Mercurial, 3.8.3, was published on the first of June 2016. On 28 April 2016, version 1.9.4 of Subversion was published. For GIT, version 2.8.4 was the last released version control system. It was released at the seventh of June 2016. These are all quite recent, and thus deserve the maximum score for this point.

Avoiding conflicts is mainly done by using clever merge strategies that can do some things already automatic. One feature that might aid to this goal is using external merge tools that are especially made for the purpose of avoiding conflicts. All three version control systems do have the possibility to plug in external merge tools, which means that for avoiding conflicts there are not many problems.

Hybrid behaviour of version control systems means that they are a combination of a centralised and a distributed approach for version control. A centralised approach for version control means that one central remote server is made, and committing new versions for storage can only be sent directly to this shared remote server. A distributed version control systems means that there is no shared server. All development is done locally on several computers indepedently. Working together is achieved by sending each other so-called patches. A patch is a set of changes that can be applied to a certain version of a model to upgrade it with the new features of the patch. The ultimate version control system for model-driven version control would be a combination of both: having one remote server that collects the overall work, while still being able to do some version storage locally and without network on a separate

[1](Chacon and Straub (2014))
[2](Collins-Sussman et al. (2011))
[3](O'Sullivan (2009))

computer. GIT is officially distributed, but does have quite some support for remote access. GIT also has built-in functionality, like " git push", "git pull", and "git fetch", that are specifically introduced to do communication with remove repositories. SVN is purely centralised. Another version control system, SVK, expands SVN with distributed features. Unfortunately, the status of development is discontinued for SVK. Mercurial on the other hand is quite distributed, but does have built-in support for remote servers just like GIT.

3.3 Desired project structure

One model itself under version control is not a problem. It becomes more difficult once this model is part of a larger directory structure. For this section, the assumption will be made that there is a project under which a folder with the name Models is created that stores all models related to that specific project. The left part of Figure 3.3 shows an example of such a project structure, in which the Models folder is collapsed. The following demands were made for this project structure:

- The projects folder contains a folder that stores all models (as can be seen in the left part of Figure 3.3).
- To avoid intertwined history, the projects folder, models folder and all models themselves should maintain their own history. Each model should maintain history about operations that only concern that model. The models folder should maintain history about operations that involve multiple models at the same time. Finally, the projects folder is for maintaining general project operations in history.
- GIT can only perform updates on a complete tree at once. This means that to only update one model at a time, the model should get its own tree.

GIT has no direct way of providing this functionality of switching only one model or switching only one folder under another folder. There is however a technique in GIT called subtree merging (chapter 7.8 of (Chacon and Straub (2014))). Subtree merging allows to create an alternative file tree next to the project root tree, which then can be hung under a folder of the original tree. The right part of Figure 3.3 shows a three-step approach of performing this subtree merging action in GIT. The first step is creating an initial project structure, in which a Project folder and the underlying Documents, Images, and Tests folders are created. By making five copies of this "master tree", alternate histories can be created for these trees from that point on. The trees can also be seperately altered. Step 2 does exactly this and alters one of the copies to contain the Models folder. Subtree merging is now exactly that: merge a tree (in this case the Models folder tree) to a subtree of another tree (in this case the Project folder tree). The final step is to alter the other four copies into trees that only have one single model. These model trees should be hung under the Models folder tree. This is done via subtree merging again.

At this point, it becomes possible to update only a single model, since it has its own tree. Any GIT action that would regularly be performed on the top-level tree, can now also be performed on these subtrees (adding models to be monitored, sending a new version, and so on). Switching between trees can be done via the usual git checkout command.

3.4 Combining models

As was discussed in the literature research chapter (chapter 2), model transformations often lie at the base of combining models. The differences between two versions of a model can be described as the set of atomic actions applied to the first model to get the second model out of it. Unfortunately, 20-sim does not support model transformations. There is no such thing as atomic actions that are tracked by the software in 20-sim. For that reason, these atomic actions have to be recreated based solely on the two versions of the model. To do so, it is crucial to have a basic layout that each 20-sim model should fulfil. In that way, basic components from 20-sim can be identified that mean something to the user, and that could be interpreted as atomic actions between the two versions of the model.

The 20-sim format is an XML file. This means that it consists out of a set of tags and attributes. An XML-file should though fulfil certain criteria to be a valid 20-sim model that can process and simulate. To make sure that these criteria are satisfied, a formal specification of the 20-sim model is needed. In XML, there are two formalisms that are common for specifying the requirements to an XML-file: XML

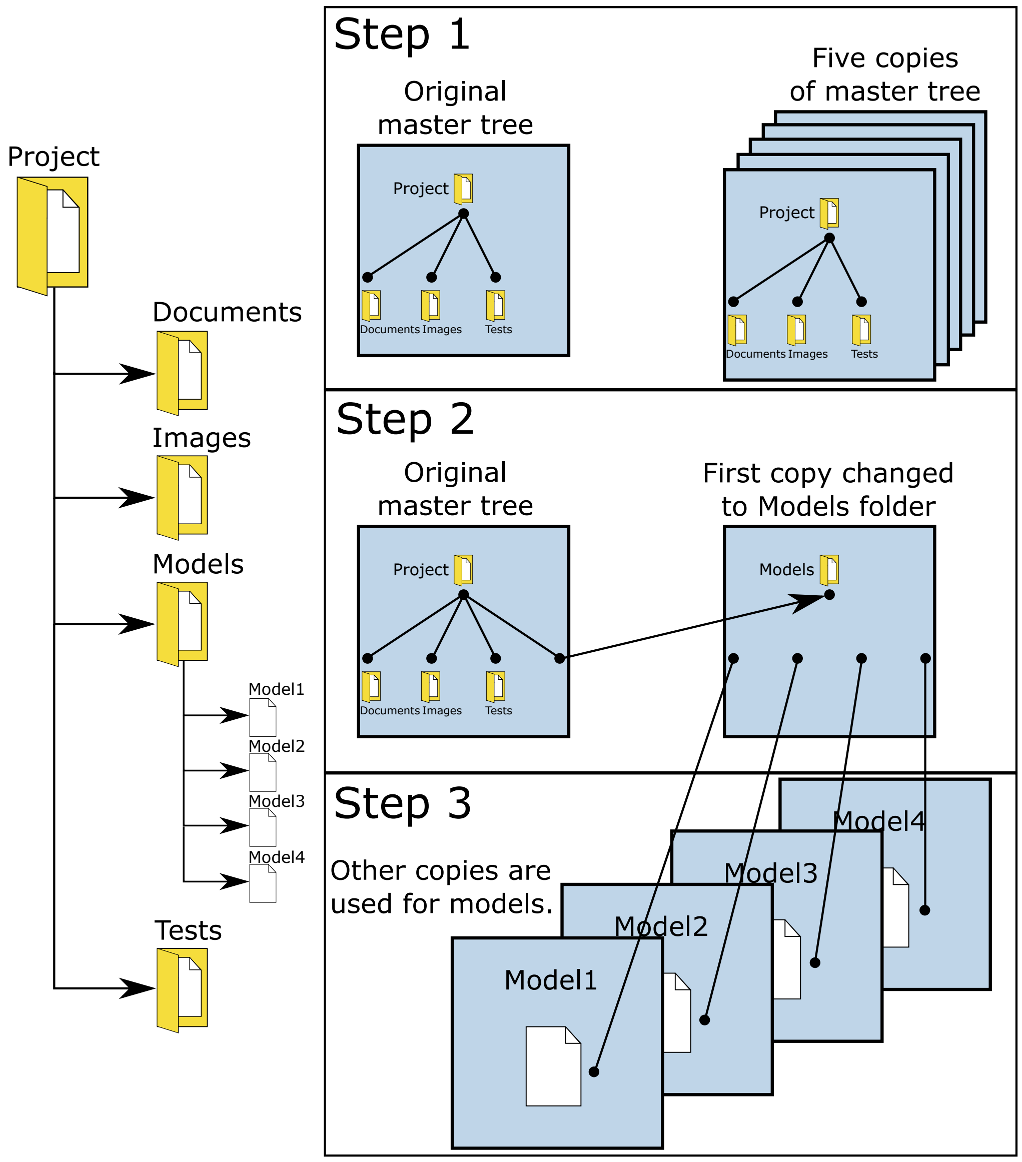


Figure 3.3: The project structure tree from the Project root folder to a Models folder with separate models underneath (left), and the approach in GIT to recreate this tree via branches (right).

Schema Design (XSD) files and Document Type Definition (DTD) files. The first one is written in XML, and thus easier to learn for someone already working in the XML language. The latter has its own syntax, and thus needs a steeper learning curve to write XML specifications in. On the other hand, DTD files are used as specification for more than only XML files (SGML, HTML, and so on). The choice for the XSD formalism was made based on the first argument about the XSD being written in XML too, in combination with the fact that Controllab Products B.V., the company at which this assignment is done, has already prior experience with XSD's. This prior experience could help in pinning down issues that arise upon using the XSD formalism.

An XSD is not enough by itself to create a valid 20-sim model though. XSD specifications cannot capture all implicit assumptions of the tool, since structures like if-then-else rules are not properly supported in XSD's. An example of this lies in integration methods. Some integration methods only need a stepsize to work in 20-sim, whereas others need much like relative tolerance, absolute error margins and relative error margins. The XSD can capture that a choice has to be made between integration methods. It can also capture that tags like the relative tolerance are optional. However, it cannot overcome the problem that if an integration method without the option for specifying a relative tolerance is selected (like Euler), that the tag for the relative tolerance can still be present under an Euler integration method. The reason here is that the XSD cannot lay this link between a choice and the consequences for the XML. Problems like this can only be overcome by defining a set of rules that handle the rest of the validation of a model. This can either be done by forcing the code that generates these tags to give an indication that a certain impossible combination has been created or by neglecting tags that are present, but that do not contribute to the model behaviour. The prior option has the advantage, since unneeded tags should not be present in the model XML where possible.

The next step would be to merge these two initial versions of the model, which usually also goes via these model transformations. The set of atomic operations is still defined as the exact difference between model version 1 and model version 2. To merge these two models, take the first version and apply the atomic operations corresponding to the desired features from model version 2. As was discussed, atomic operations are not supported in 20-sim. However, it is possible to obtain that part of the 20-sim XML that corresponds to a certain user-intuitive 20-sim component like "Submodel" or "Plot". It then becomes possible to compare both versions of the model just like model transformations usually would do. In section 2.6, the set of basic actions for model transformations were already mentioned as being insertion, deletion, and modification. These three actions can still be applied to the difference and merge system described in this section. It can be checked if a component is present in one but not in the other version (insertion or deletion), and it can be checked if there is a difference between a component that is present in both versions.

It is not too difficult to combine two XML files, it becomes difficult however if the resulting XML file should also fulfil the specification of a 20-sim model that can be processed and simulated. This is however exactly what is needed for this step of combining two models. Even though the XSD might not always be as complete as would be ideal, it still contains a complete set of all XML tags that could be present in the model XML. If the tag is not in the XSD, it can never be present in the XML. The idea here is thus to start off with building a tool that reads the xsd, and assigns the corresponding xml tags along the way. Eventually, two such "XSD trees" will be made (for each of the two models one tree), which then can be compared and checked for differences. This will be further elaborated in section 4.4, including a picture on how this is done in the application designed in this master assignment.

The next step in combining these two models is to use this newly gained data structure, and obtain components that mean something to the modelling tool. Things like connections, plot objects or submodels are much more intuitive to the 20-sim user, than random tags from an XML file. Therefore the xml tree will be cut into pieces and assigned to the specific type of object that is to be tested. In that way, the list of differences will not be XML tags, but actual components that mean something to the user. In section 4.5, more information about assigning context is given, including a picture that shows which contextual blocks were taken for the proof of concept and how they relate to each other.

Finally, the user should be made aware of the differences between both versions of the model. The extracted 20-sim components (submodel, plot, etc.) taken from both XML representations of the 20-sim models should be used to show the insertion, deletion, and modification in a graphical manner.

3.5 Impact

A difference tool for 20-sim is one interesting feature to add. However, the functionality that such a tool can offer can be much more than just viewing differences and merging models. During this master assignment, part of the time was focussed to look at the impact that these differences had, mainly on the simulation results of a 20-sim model. This section will discuss exactly that: strategies for finding the impact of differences between versions of a model.

In section 2.6, the idea was introduced to use so-called traces to find relations between the model itself and the surrounding concepts of that model like requirements, results of simulation and unit tests. This gives the need to determine when a model has changed in such a way that it affects those requirements, simulation results and unit tests. The proof of concept built in this master assignment is used to determine if a submodel has been "operationally changed". This "operationally changed" means that by applying this difference, the simulation results will change. For example: a change in the equations is an operational change, but a change in the name or color of a submodel is no operational change. This "operationally changed" concept is heavily based on including semantics in the model comparison, as was suggested by Reiter et al. (2007). Reiter et al. (2007) already described that semantics can have large impact on the model behaviour. Because the tool provides a level of context (see section 3.4), it becomes possible to invalidate results generated by a certain contextual component. For example, if a submodel was used to generate C-code with via the real-time toolbox of 20-sim, it will become outdated C-code if the submodel that generated it got operationally changed. Similar things can be done for acceptance tests, unit tests, resulting plots, requirements or documentation. This idea thus can give the user feedback on automatic regeneration of quite some related functionality of that specific contextual component, in this case a submodel. section 4.6 will go more into details on how this was implemented in the first proof of concept.

Another aspect of impact, is that a list of differences is just a plain, unsorted list of differences. The question is which difference should get priority for attention of the user? The user does not always want to go through all minor differences trying to find the perfect merge, but sometimes likes to do a quick merge of all major differences. In that case, a notion of a major difference should be defined, according to the user its wishes. A first estimate is that the user wants to see those differences that impact the simulation results the most. By using artificial intelligence, it becomes possible to generate a set of learning examples that focusses on impact on these simulation results. Another advantage of using an approach from the field of artificial intelligence, is that the user can add his own learning examples if he does not agree with what differences are labelled to be major. The technique used in this report is based on Bayesian learning, which is explained in more depth in Appendix D. The idea of Bayesian learning is that, based on a set of statements that can be either true or false, an estimation can be given of this impact score. Thus for a submodel, some statements could be: Submodel has been operationally modified, submodel has many ports, and submodel has newly added implementations. Given these statements, and what the impact score was for previous learning examples with specific combinations of true and false for these statements, it becomes possible to do an estimation of which difference is the most relevant to the user. The algorithm to do this, is also explained in Appendix D.

3.6 Conclusions

The design chapter indicated that two main points are interesting for investigation in the implementations chapter.The first being version control and how an existing version control can be used to implement a model-driven version control system in which each model can be separately monitored, updated and tracked over history. The second main point was the fact that a proper tool needs to be developed to find differences between two model versions, compute the impact that these differences have on simulation results, and merge the models in such a way that the resulting model can be opened and processed in 20-sim without any modifications. Especially the latter point will be worked out in the implementation chapter below.

Chapter 4

Implementation

This chapter presents the complete overview of the implementation of the tool. It will start off with the initial design of the version control system in section 4.1. The other sections are all about the main application of this master assignment: the comparison tool. The flowchart shown in the Design chapter in Figure 3.1 has been adapted to include the actual flow of the comparison tool, as shown in Figure 4.1. Note that the Model Management and Version Control System blocks are minimised, to give more space for the Tool block. This figure will be the base for the rest of this chapter, because it will be used to place each section in the context of the comparison tool or the overall picture of model management.

4.1 Version control

Version control lies at the base of model management. It is therefore the starting point of the flowchart of the comparison tool. In fact, version control will call the comparison tool for differences (and a merge) when needed. Version control is indicated by the letter A in Figure 4.1. An initial attempt was made to use version control to make a project structure that adheres to the specification given in Figure 3.3. The technique used for this is called subtree merging. Subtree merging works with creating additional so-called branches of a model. A branch is a way of creating an alternate timeline in GIT. Branches are common in GIT for trying something out, and merging the results back to the original master branch. Effectively however, a branch is nothing more than a tree.

For subtree merging, the main master branch is corresponding to the top-level Project folder in Figure 3.3. For the Models folder, an empty branch is created, which is in fact a copy of the Projects master branch that is emptied afterwards. This now empty tree can be used to add a Models folder, and later on the models underneath. Subtree merging is the technique to connect this tree under the project folder in the original master branch. For the separate models in the Models folder, the same thing applies. Each model gets a separate empty branch to which the specific model is added. These model branches are added under the Models folder branch. At this point, the tree structure in Figure 3.3 has been recreated.

Branches have their own history, such that this history is available at project level, Models folder level, and at separate model level. Switching in between these levels in the project structure can be done via the usual command for switching between branches, namely the checkout command of GIT. Furthermore, the general commands of GIT can be used to do the basic version control operations of file-based version control on models. For example, the commands "git add" and "git commit" can still be used to respectively label a file to be monitored in the version control system and storing a version of the changed files that should be monitored by the version control system.

Besides the project structure, another challenge with GIT was found. GIT namely only shows one version of a file at a time. Under regular usage in GIT, it is not possible to see two versions of the same file next to each other, let alone compare them. The most simple solution to this problem is to first obtain a version of the model, copy it to another temporary location, then switch to another version of the model, and compare the current version with the just copied version. Another solution is to copy the branch of the specific model, and paste it next to the current model by using subtree merging. These two branches are now in fact identical, and thus one branch can be placed back to an earlier version of the model. Comparing both branches now has the same effect as comparing an old version of a model with a newer version. All GIT-related actions that have to be taken to build the project structure, to

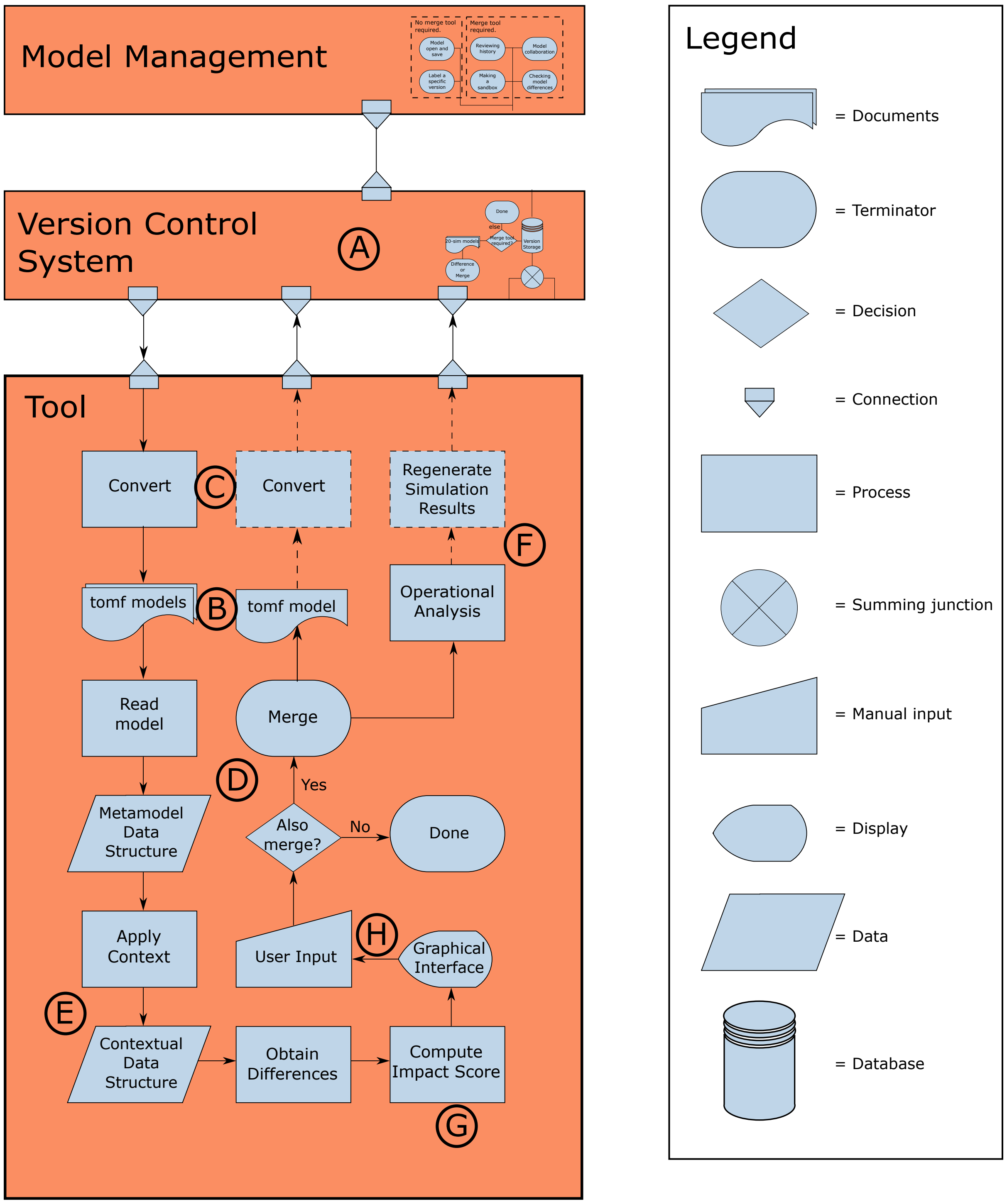


Figure 4.1: The flowchart of the Implementation of the model comparison tool. The Model Management and Version Control System blocks are not that relevant to this chapter, and are minimised. The flow of the application to be developed is described in the "Tool" block.

manage the versions of a model and to show several versions of a model next to each other are discussed in Appendix A in more detail.

4.2 Metamodel development

Before starting with the actual tool, the decision was made to create a new model-format for 20-sim. The original model-format of 20-sim is the so-called Expanded Model Xml (EMX) model format. This model format is one single file that is half XML, and half plain text. The XML part of this format describes the plotting section of 20-sim, and a lot of toolboxes. The textual part contains information about anything related to the 20-sim canvas in which the actual model is built. Both parts give essential information about differences between two versions of a model, however plain text is very difficult to interpret. It becomes even more difficult because of the fact that part of that plain text is written by the user. The text as the user types it in 20-sim equation models, is inserted in this plain text. The user is in that sense completely free to add as much tabs or spaces as the user desires, and one equation could be spread accross multiple lines. All these challenges with plain text in the model file were the first reason to design a new format for 20-sim. The new format should be completely in XML, such that it is easy to interpret. To do so, the existing knowledge that was already present in 20-sim, was used to write the full information about the canvas to XML format.

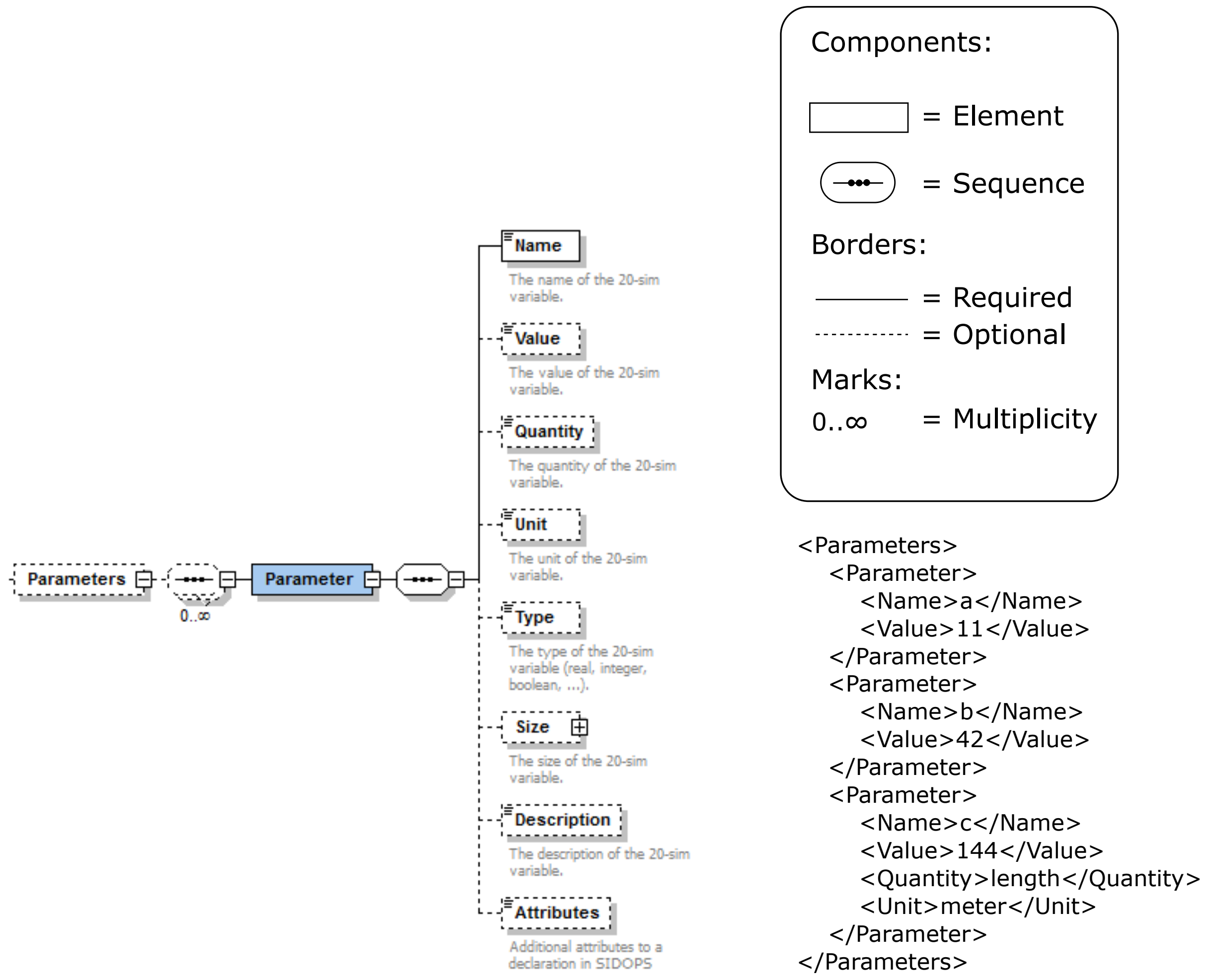
Another reason to design a new format for 20-sim, was the fact that the format was frequently expanded over time when new features of 20-sim were added for storage to this format. This growth of the format had the consequence that due to reasons of backward-compatibility with older versions of 20-sim, new information often was stored at positions that did not make sense from a tool perspective. One such an example is the relation between plotting windows and the plots they contain. One plotting window can contain multiple plots in 20-sim. These are however stored in the EMX format as completely separate tags, linked via a set of plot IDs. What would make more sense is to store the plots as actual childs of the plotting window they are part of.

The two design criteria mentioned above, about the conversion from plain text to XML and the more logical ordering of 20-sim components, resulted in a new format called the 20-sim Open Model Format (tomf). The new format is shown in Figure 4.1 at the letter B. The idea behind this new format is that it consists out of three things: the model file, a metamodel file, and a set of implicit rules. 20-sim has default values for almost anything. Take for example the Euler integration method with step size of 0.01 seconds. This is the default choice in 20-sim to perform a simulation with. If a user would not change this, then it is not relevant to store this in the model file, since it is default behaviour. It starts to become relevant to store once the stepsize or integration method is changed by the user to another integration method. On the other hand, there should be something that does know this value. How does 20-sim know the default behaviour, if it is nowhere written down? This is where the second part of the new format comes into play: the metamodel file. The metamodel stores what the default behaviour is of 20-sim, what information is stored at what position in the model file, and how many times information can occur. An example of the latter criterion is for example that it is possible to have 20 plots in a plotting window in 20-sim, but a plot can only have one type, like a graphplot or a 3D animation plot. Finally, the set of implicit rules indicates model behaviour that cannot be captured in the metamodel. These rules usually are if-then relations. An example of such a rule is the fact that if a variable has the quantity "current", then its unit cannot be "metres".

The metamodel used in this report, as was described in section 3.4, is called an XML Schema Design (XSD). To get an idea on how this works, an example was made for the case of Parameters. This example is shown in Figure 4.2. Basically, there are two components here: an element and a sequence. An element in the XSD directly maps to a tag in the XML, and a sequence shows that the elements (or tags) underneath it should occur in that order in the XML. The multiplicity indicates that under the Parameters tag in the XML there can be any number of Parameter tags. Furthermore, every Parameter tag needs a Name underneath it, but all the other tags are optional. The right part shows a piece of XML that satisfies this schema. For more information about the supported constructs of this XSD format in the comparison tool, take a look at Appendix B.

4.3 Conversion to the new format

With the new format in place, this section will go into more detail on the conversion from the currently known model format in 20-sim (EMX) to the new format (tomf). The flowchart includes the conversion


Components:
= Element
= Sequence
Borders:
= Required
= Optional
Marks:
0..∞ = Multiplicity
Parameters
0..∞
Parameter
Name
The name of the 20-sim variable.
Value
The value of the 20-sim variable.
Quantity
The quantity of the 20-sim variable.
Unit
The unit of the 20-sim variable.
Type
The type of the 20-sim variable (real, integer, boolean, ...).
Size
The size of the 20-sim variable.
Description
The description of the 20-sim variable.
Attributes
Additional attributes to a declaration in SIDOPS

```xml
<Parameters>
   <Parameter>
      <Name>a</Name>
      <Value>11</Value>
   </Parameter>
   <Parameter>
      <Name>b</Name>
      <Value>42</Value>
   </Parameter>
   <Parameter>
      <Name>c</Name>
      <Value>144</Value>
      <Quantity>length</Quantity>
      <Unit>meter</Unit>
   </Parameter>
</Parameters>
```

Figure 4.2: On the left a visual representation of part of the XSD model, made in Altova XMLSpy 2016 Professional XML Editor. On the right a piece of XML that satisfies this XSD model.

from any EMX model to the new tomf format, as shown by the letter C in Figure 4.1. The conversion process itself is shown in Figure 4.3. Note that the EMXConverter is the C++ application written for the sole purpose of converting from the EMX format to the tomf format. There are two parts of the EMX format that have to be converted to the new format: the grapheditor and the simulator. The grapheditor contains information about the submodels, their connections, their parameters and their equations. The simulator contains all information about performing simulations, including the plots, toolboxes and integration method information. As can be observed from Figure 4.3, the simulator part of the EMX format is already in XML. This conversion can be done straightforwardly by rearranging the tags in the XML such that they fit the new metamodel format. This conversion is performed by using the XML-parser TinyXml-2 (see Appendix C for more information about TinyXml-2). This parser first creates a tree from the XML part of the EMX format, and then creates a new tree in which it places the tags in the proper order and with the proper name for the new format.

The grapheditor part of the EMX model cannot direcly be converted to the new format. Instead, 20-sim is used to generate this part of the XML directly from the processed information obtained by 20-sim. To do so, a call is made via the scripting interface of 20-sim to the XML code generation functionality of 20-sim. In there, a specific XML code generation target was added that is used to generate this part of the XML for the new format. The call to 20-sim opens the 20-sim EMX model, processes the model to obtain all information needed about the grapheditor, and then generates the new XML code. This XML code is then via the EMXConverter application read in C++ and added to the overall C++ tree for the new format.

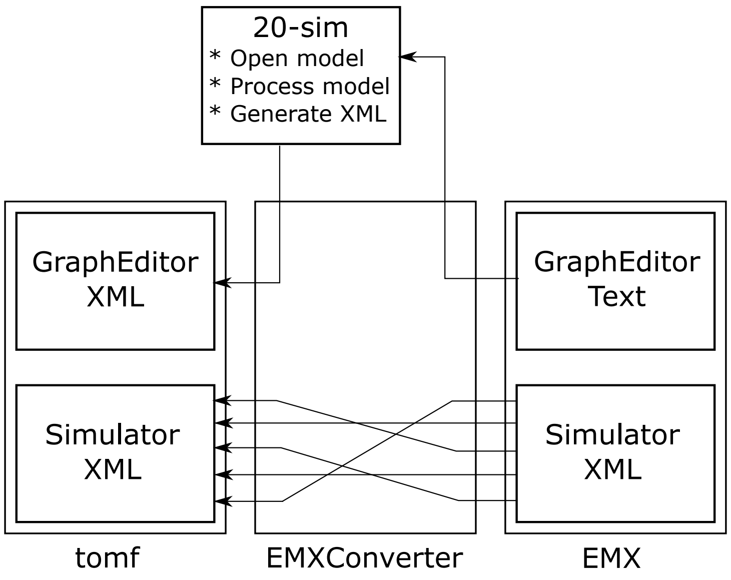


Figure 4.3: The conversion process between the old and the new format.

4.4 Metamodel-based model comparison

Given the fact that an EMX model is now in the new format, the next step is to read in this new format in such a way that it can be compared to other models in the new format. To do so, the metamodel of section 4.2 is used to make a tree that contains all information that is needed for 20-sim to open, process and simulate a 20-sim model. Take Figure 4.2 as an example. The XML has the information about three parameters, their names, their values, and for the last parameter also the quantity and unit. The metamodel knows that there can be other optional tags like Description, Size and Type. It also knows that the default value for a parameter is that it has no description, a size of 1 by 1 (scalar), and that the type is by default a real number. To include all this information of both the XSD and the XML, a tree is made in a C++ application that runs through the entire XSD metamodel. For the specific part of the XSD corresponding to the parameter example in Figure 4.2, this process has been worked out in Figure 4.4. Every block (both dashed and solid lines, except for the legend) is an actual instance of a class made in C++. These instances are either of the class Element (solid lines) representing a tag in the XML or off the type Sequence (dashed lines), forcing its child elements to be in that exact order in the XML. The left side of this figure shows exactly the Parameters example of Figure 4.2. The tag highest in the hierarchy is the Parameters tag. Underneath, there are three sequences that each contain a Parameter tag and inside this Parameter tag another sequence. Underneath this sequence, there are the actual properties of the parameter. On the left side of the figure, these are the parameters a, b, and c with respectively the values 11, 42 and 144. The right side of the figure shows another version of this model, in which the parameters a, b and c are adapted. Parameter a now is an integer number, parameter b has another value and parameter c has no more a quantity and unit.

At this point, a major breakthrough has been reached. In section 3.4 it was already shown that model transformations are usually the way to go. This same section also discussed that 20-sim does not monitor these model transformations, but that the desire still exists to see if there is a possibility to detect these atomic actions. These atomic actions can now be found with the results just shown. The XSD metamodel makes sure that every tag, even the optional ones, are included in the tree. The model file assigns its information to the corresponding XSD elements in that same tree. Because the metamodel is the same for each model, it becomes possible to start to find differences between tags in two trees under comparison. To illustrate this, take another look at Figure 4.4. Based on the schematic of the XSD metamodel in Figure 4.2 it becomes clear that underneath a Parameters tag, there is always a sequence that is followed by a Parameter tag. Furthermore, under a Parameter tag, there is always a set of properties of that parameter. Figure 4.4 shows in both trees all these XSD metamodel components. Because this structure is part of the XSD metamodel, it is known beforehand. At the point that a Parameters tag is found at exactly the same place in both model files (under the same equation model for example), it becomes possible to start to go along the XSD Parameters structure, and compare both files. In this case, differences can be found in parameter a its type, parameter b its value, and parameter

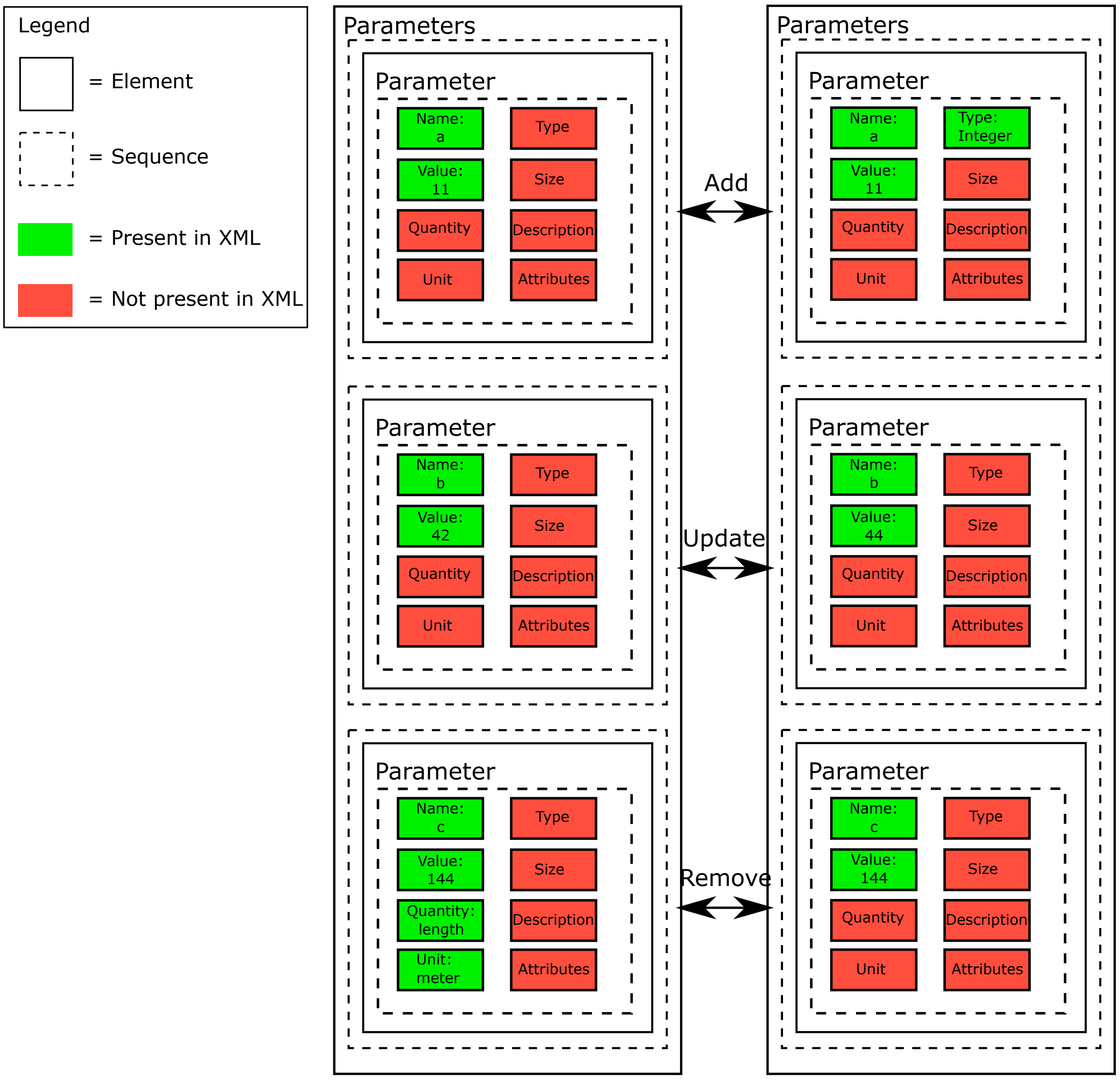


Figure 4.4: Reading in the XSD tree, based on Figure 4.2 (left) and comparing with a slightly modified version (right).

c its quantity and unit. Figure 4.4 also indicates these differences with three types: add, update and delete. The same model transformation types were mentioned by Taentzer et al. (2014) and Brosch et al. (2010).

Note that the XSD metamodel is crucial here, because this means that the exact model structure is known before the comparison. This becomes even more relevant when considering a merge of these two trees. In case of Figure 4.4 it is known that the merge would result in a Parameters tag, with underneath three times a parameter tag. In these Parameter tags, the chosen merge strategy is performed. Take for example Parameter b. There are two options here: merge parameter b with value 42 or with value 44. The final merged result thus contains a parameter b that has either value 42 or value 44. If no XSD metamodel would be present however, a common mistake in merging can be made: the Value tag can be present twice in the merged result. Merging two XML files does not protect against this, since the XML has no knowledge about what is a valid model or not. By using the XSD-XML trees, this restriction can be placed. The flowchart of Figure 4.1 also shows the letter D to indicate all functionality that is coupled to this phase of the implementation of the comparison tool. Not only is this tool useful to read in the new format in a tree structure, it is also used as the base for the merge.

4.5 Assigning context

The previous sections were just working with XML tags. These tags are not very easy to interpret by users. Therefore, an additional layer of context was developed. This context should provide a wrapper around the XML, such that the differences between the models should be in intuitive concepts like a plot or submodel. The flowchart of Figure 4.1 shows this step in the process of comparing two models with the letter E. The set of currently supported context blocks is shown in Figure 4.5. Note that all classes in this diagram share a common parent class that defines the set of minimal functions to implement for each context class. This parent class is called the Block class. It forces each child class to have at least the following functionality:

1. A function that writes the XML tree corresponding to the current contextual block to file.
2. A function that compares two blocks and checks if they are equal to each other.
3. A function that gathers a list of all differences between two block instances.

Besides this functionality, there is still a function planned to be added that will be able to write user-intuitive messages to a user interface about what the actual differences are. One can imagine that only seeing that a curve in a plot has changed is useless without knowing the details about what has changed to the curve. An example of such a message can be: "The curve changed colour from blue to green".

The Block class childs in Figure 4.5 start out with a Model on top. This Model is split in a GraphEditor part and a Simulator part. The GraphEditor represents the root model in the 20-sim canvas, and thus can have any amount of implementations, represented by the Implementation class. Implementations can have any number of interface ports and interface parameters, and can be either graphical or equational. Furthermore, equational models can contain parameters, and graphical models can contain again a set of submodels, connected by connections. On the Simulator side there are parameters that can be changed in the simulator. These are parameters that are already defined in the grapheditor part, but for example changed unit in the simulator for plotting purposes (the plot should be in miles, but the computations were done in kilometres in the model). Furthermore, the simulator contains user-settings for various dialogues (including the simulation method and step size) and toolboxes, and it contains plotting windows called plotpanels, followed by the plots that are stored within them. These plots can be of various types, but the objects they represent are either curves or 3D objects. The division described above suggests that it is currently only possible to show differences at context block level, since other properties that are not given their own Block class are still only present in the XSD-XML tree. Some of this problem is solved by displaying additional information in the graphical user interface, as will be discussed in that section (section 4.8).

A challenge that arises with these contextual blocks is how to handle things like timestamps. There is almost always a difference in timestamp, but this is not a difference that should pop-up in the comparison of two models. There are many more of these types of differences that are actually not relevant to account for. The tool supports filters that are specific to the context class, that filter out any tag that should not be compared for upon looking for differences in that specific context class.

4.6 Finding operational differences in submodels

At this point, differences can be found between two model versions. However, currently every difference is the same, from a difference in font type of the label of a curve in a plot to the fact that there might be actual equational changes in a submodel. The latter is much more relevant to see for a user, since it actually changes the behaviour of the simulation. The tool however does not have this notion of relevance yet. This section will describe a first approach on finding submodels that have been operationally changed, because their equations do not represent the same input-output relations anymore. The step of operationally finding differences is part of the flowchart in Figure 4.1, and is shown by the letter F.

To obtain a set of equations of the overall model in 20-sim, one can perform a processing action. The result of this processing action is a set of equations that is simplified and which is already verified to be valid. These equations were used to make C++ constructs that look the most like the concept of a C++ stack (cplusplus.com (2016) describes the definition of a C++ stack). Although in contrast to regular C++ stacks, the stack-like objects used here are First-in First-out (FIFO), whereas stacks are Last-in First-out (LIFO). Also, the stack-like objects in this application can be resetted. This means

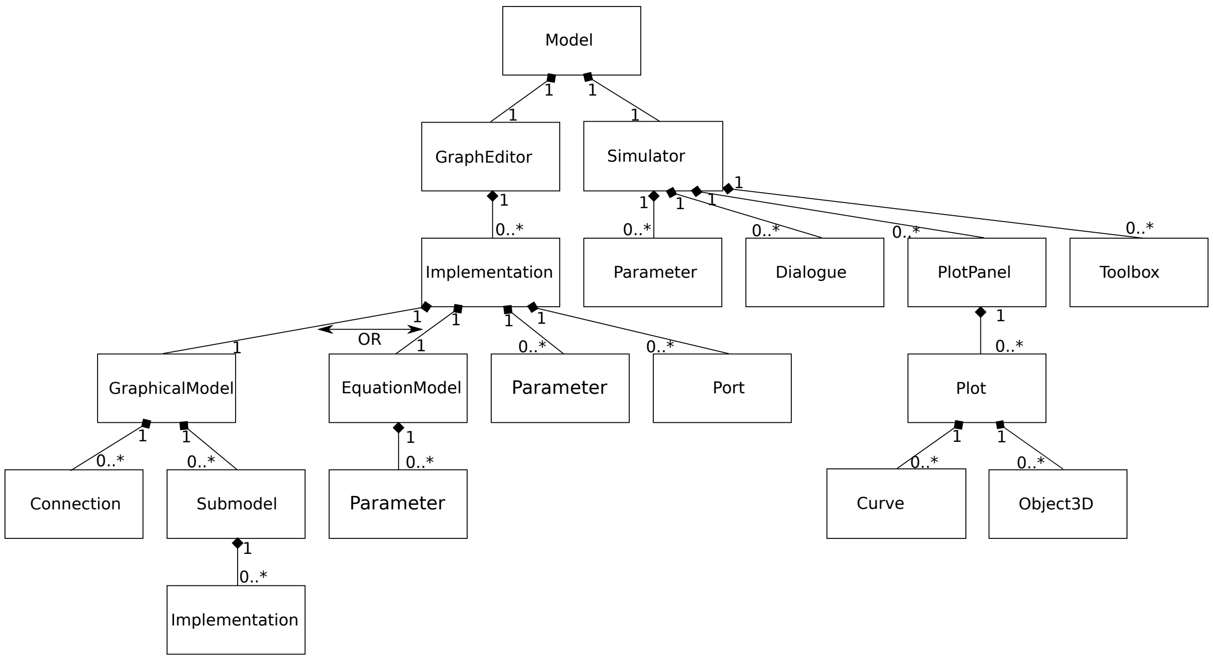


Figure 4.5: A UML diagram of all contextual block classes for a 20-sim model, that were used during the scope of this master assignment. One such a class contains part of the XSD-XML tree that is relevant for that contextual block, and it has functionality for writing to file, checking for equality of two context blocks, and obtaining a list of differences. All classes in this diagram are childs of a common interface parent class called Block.

that objects do not disappear once taken from the stack, but a simple pointer is incremented to go to the next object, while maintaining the previous object.

Besides these differences with the original concept of a stack, the class was still called ImpactStack. The reason for this is the fact that the operations push and pop were used as the basic interaction with the stack, and (an adaptation of) the reverse polish notation (as explained by McIlroy (2016)) was used as a base of reading operators from the stack. The idea is to use the operator first, and then mention the arguments. For example a+b becomes +(a,b). There were two reasons for choosing this approach over more conventional algebraic approaches. The first is that at the moment that an operator is read, like the + operator, the application knows that it can expect two arguments to come for that specific operator. Using a + b does not know this, until the + operator is read. This means that a variable "a" is found, without knowing of which operation it will eventually become part. The second reason for using this notation, is that these equations are also part of the new format of 20-sim. It is very difficult to represent a + b in XML in such a way that the operator can be easily taken out of the XML. One such an implementation would look like:

```
<equation>
        <variable>a</variable>
        <operator>+</operator>
        <variable>b</variable>
</equation>
```

One thing to note is that there is apparently no hierarchy between the variables and the operator that is working on those variables. The whole equation has to be read up to the + operator, before it becomes clear what operation is performed. A better way would be to represent it like this:

```
<equation>
        <operator name="+">
                <variable>a</variable>
                <variable>b</variable>
        </operator>
</equation>
```

This is exactly the result of the discussion held in this section so far, about how to represent equations in a stack-like manner. All these ImpactStack objects exist out of three main components: Literal, Variable, and Operator. These are three C++ classes, each having one shared parent class called Stack-

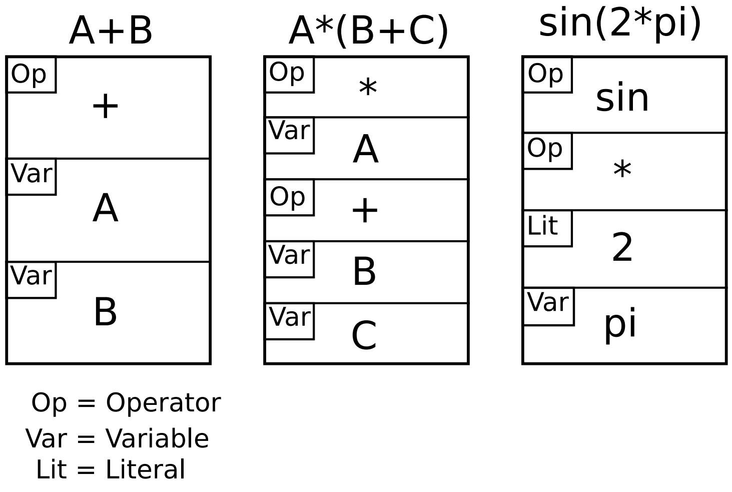


Figure 4.6: Three examples of visual representations of ImpactStacks. On top of the ImpactStack there is the equation it represents. The top left corner of each Operator, Literal or Variable shows which one of these three types it is (see legend at bottom left of figure).

Element. Some examples of these impact stacks are shown in Figure 4.6. The most left one of the three is the one already shown as an example in this section before, namely a+b. The idea is to start reading from top to bottom in this figure. First the + operator is found, then the first argument, namely variable A, and then the second argument of the + operator, namely variable B. The second stack shows a nested operator impactstack. The operator with most priority is the * operator, which is shown on top of the impactstack. Its first argument is the variable A, and its second argument is the + operator. For the + operator, the first argument is the variable B, and the second argument is the variable C. Finally, the third impactstack shows that functions can be interpreted as operators, since they are operators with one argument, and the actual behaviour of the function does not matter for this class (there is no actual computation of the sine of 2*pi, it only notes that there is a function called sin that needs one input argument). The first, and only, argument of the sin function is the * operator, which needs two arguments. The first argument is the literal 2, and its second argument is the variable pi. Note that pi usually is not seen as a variable, but since literals are in the generated equations of 20-sim defined as only numbers, it is a variable in this case.

Another interesting thing is that variable names have no meaning in this application. The only concern is to check if the input-output relation of a submodel is the same. If a model thus contains the variable Q, which is equated to the current simulation time in 20-sim, then it does not matter that this variable is later on renamed to P. Even if in the first case: output = Q * input, and in the second case: output = P * input, the generated equations between input and output have not changed. Another feature that is supported is the mathematical rule of commutativity. This states that if an operator is commutative, the following should hold: $operator(a, b) == operator(b, a)$, which simply states that interchanging the input arguments should not change the outcome. An example is a+b, which is the same as b+a, and thus + is commutative. If an operator is commutative, the tool knows that it does not matter which input argument is offered first to the operator, and thus it does not detect an operational difference if one of the models contains a + b and the other contains b + a. One could of course extend this behaviour with rules of associativity, and include matrix and vector operations (for example, the * operator is different for vectors and matrices). At this point, the .* (elementwise multiplication) is set to be commutative, but the * operator is always non-commutative, since no difference is made between scalars and vectors/matrices yet.

Note that the previous choice for setting the * operator to be always non-commutative shows that the aim of this application is to never give false negatives. This means that every operational difference should be in the list of submodels that have operationally changed. It is however possible that not operational changes are listed as operational change. The reason for this aim is that it is much worse that an operational difference slips through the application, than that a false positive is given to the user that later on seems to not be an operational change. Especially when coupled to the idea of section 2.6, in which code, test results and much more are invalidated once an operational change is found. This would mean that if a false negative would occur, that the user relies on code that is already outdated. This should be avoided whenever possible.

4.7 Computing an impact score

Continuing the line of reasoning from the previous section, finding the impact on simulation results of changes in the model is further elaborated on in this section. The concept of using artificial intelligence in the form of Bayesian learning was already touched upon in section 3.5. The idea behind this, is that a list of differences is sorted based on what the user would like to see as a difference with high impact. This impact ranking is shown in the flowchart of Figure 4.1 as the step directly above the letter G. An initial set of learning examples is provided to this part of the application, that is an estimate based on what changes would cause high impact on the simulation results. The factors that are used as an initial guess for determining this impact score are:

- The result of the operational analysis of the previous section. If an equational submodel has changed behaviour, then it is likely that the simulation results will also change behaviour.
- The fact that ports are added or deleted to a submodel. Port changes indicate that a submodel was altered between two versions of a model. Besides that, ports indicate that the amount of input-output relations change, which might cause different simulation results.
- The fact that parameters were added, changed or deleted. Parameters are often the tunable factors in a model. Adding or changing a parameter indicates that it is likely that the behaviour of the model changes.
- The fact that new implementations were added or implementations were removed. Additional implementations or removed implementations are differences that a user likely wants to review, because they change the behaviour of a submodel completely. This in turn can have effect on the outcomes of a simulation.

These four factors are so-called known variables. There are a lot of unknown variables too. These unknown variables, or latent variables, are other factors that might influence the simulation results when they are altered. The idea of Bayesian learning is to use the known variables, and based on the learning examples estimate what the impact score is that the user is likely wanting to see. Because there are latent variables, the impact score cannot be determined exactly. There is always a probability assigned to an impact score about the likelihood that that impact score is the one the user wants to see, based on the four factors above. What however can be done, is monitoring if the user agrees with the given impact score. If the user does not agree, it is possible to take the impact score for the relevance of a submodel, in combination with the results of the four factors described above, and add it as a new learning example. This learning example then makes sure that it is weighted in the computation the next time the user sees a similar situation. The actual computation and the Bayesian learning algorithm that are used for this proof of concept are explained in Appendix D.

4.8 Graphical user interface

Since this tool might eventually be used for commercial usage, and is also used for a demonstration at the end of the master assignment, it should have a clear and easy to use user interface. The graphical user interface interaction is shown in the flowchart of Figure 4.1 as the letter H. The idea is to open a model in 20-sim, and from the Tools menu open the Model Compare Tool on this model. This process is shown in Figure 4.7.

From that point on, a second model can be chosen to compare with. The user is given the option to also perform a merge at the same time, which can be either to import elements from the selected model in the currently open 20-sim model or to merge both models together into a new model. The resulting initial window that pops up upon calling the tool is shown in Figure 4.8.

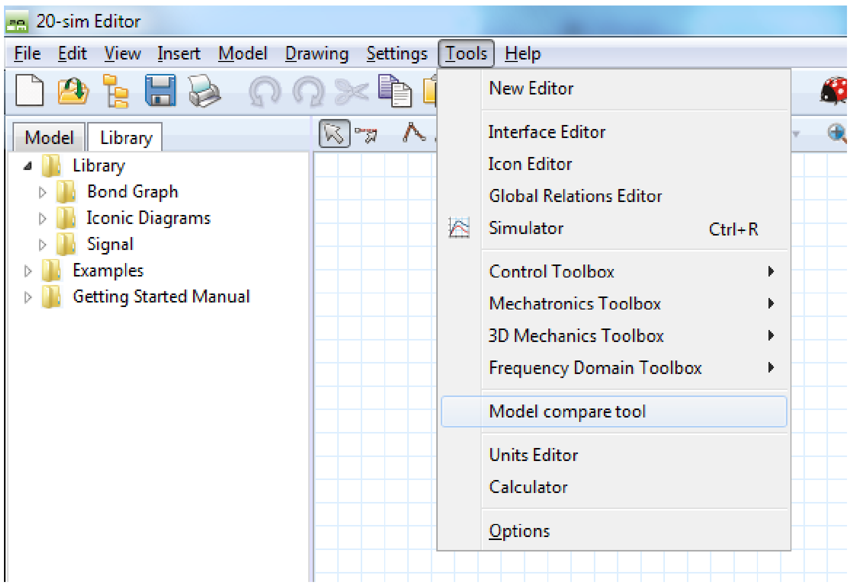
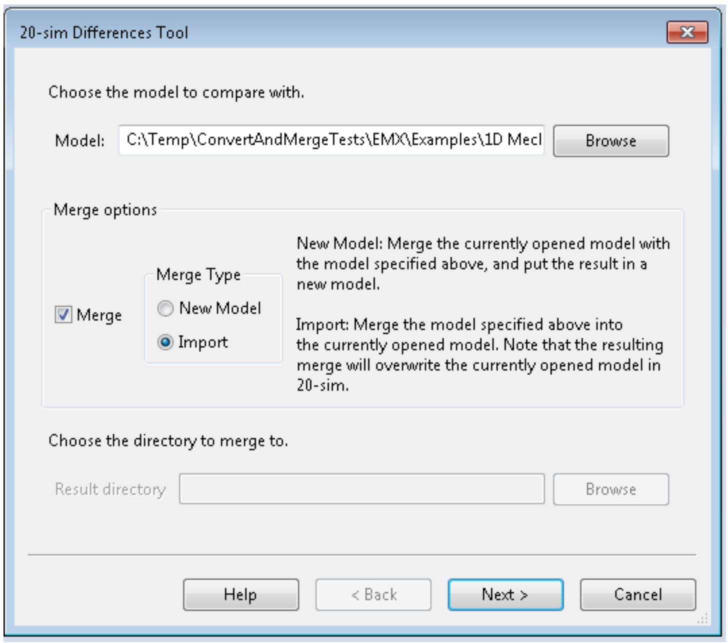


Figure 4.7: How to open the comparison tool from 20-sim.

.


20-sim Differences Tool

Choose the model to compare with.

Model: C:\Temp\ConvertAndMergeTests\EMX\Examples\1D Mecl

Merge options

Merge [✓]

Merge Type: New Model [✓ Import]

New Model: Merge the currently opened model with the model specified above, and put the result in a new model.

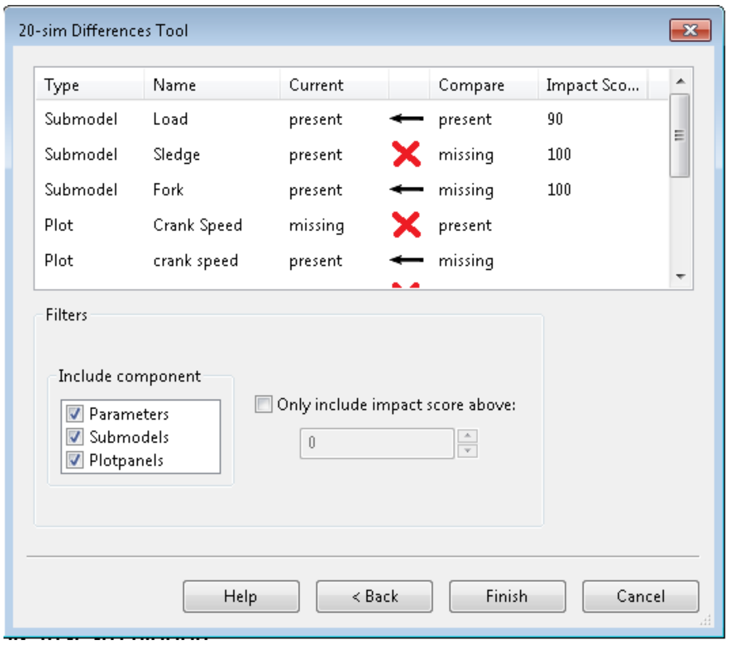
Import: Merge the model specified above into the currently opened model. Note that the resulting merge will overwrite the currently opened model in 20-sim.

Choose the directory to merge to.

Result directory

Figure 4.8: The design of the initial GUI screen for the demonstration.

The next step is to give the user a list of differences in an intuitive manner. The design is shown in Figure 4.9. The first column of the list indicates the type of change, in the figure there are changes at Submodel and Plot level. Further down the list, as can be seen in Figure 4.10, there are also differences of type PlotPanel. The second column is the name of the changed object. The changed submodels thus have the names Load, Sledge and Fork, whereas the changed Plotpanel objects are called Window 3, Crank Rod - Plots and 3D Animation. The Current and Comparison columns are respectively the currently opened model, and the model selected on the dialogue page shown in Figure 4.8. They show at this point if an object is present or missing in that version. The column with the red crosses or the arrows indicates the direction of the merge. The current settings in this list are set to be an import merge in the currently opened 20-sim model. This means that if an arrow is shown, that then the settings from the Comparison column will overwrite the settings of the currently opened model column after the merge. In case of a red cross, the settings of the currently opened model in the Current column will remain the same. The last column indicates the impact score. Anything that is not present at one model version, and that is present at the other model version has an impact score of 100%. The other impact scores are based on the four factors as described in section 4.7. Note that this dialogue also contains a list of filters, that when unchecked will remove the corresponding objects from the list. There is also a spin control for the impact score, which when enabled by the check button "Only include impact score above:" will show only items in the list with an impact score above the indicated value.



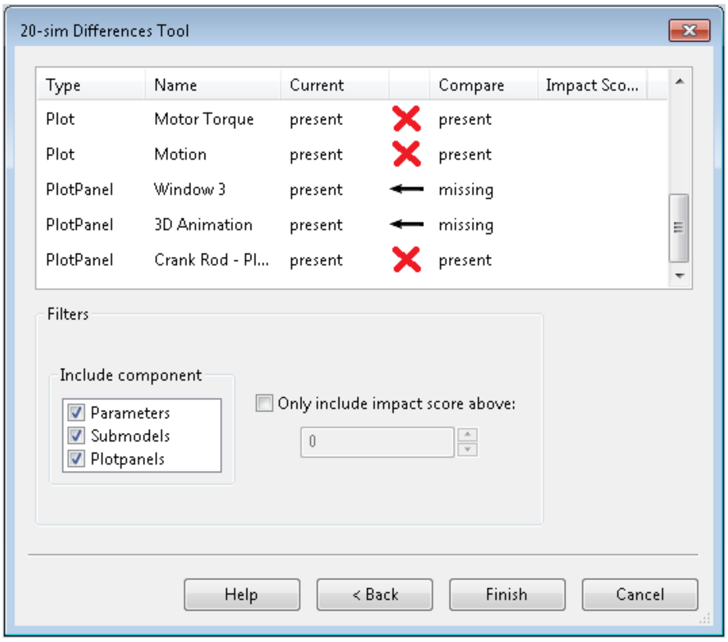
Type	Name	Current		Compare	Impact Sco...
Submodel	Load	present	←	present	90
Submodel	Sledge	present	✗	missing	100
Submodel	Fork	present	←	missing	100
Plot	Crank Speed	missing	✗	present	
Plot	crank speed	present	←	missing	

Filters

Include component
- [✓] Parameters
- [✓] Submodels
- [✓] Plotpanels

[ ] Only include impact score above: 0

Figure 4.9: The list of differences as shown by the current version of the tool.



Type	Name	Current		Compare	Impact Sco...
Plot	Motor Torque	present	✗	present	
Plot	Motion	present	✗	present	
PlotPanel	Window 3	present	←	missing	
PlotPanel	3D Animation	present	←	missing	
PlotPanel	Crank Rod - Pl...	present	✗	present	

Filters

Include component
- [✓] Parameters
- [✓] Submodels
- [✓] Plotpanels

[ ] Only include impact score above: 0

Figure 4.10: The bottom of the list of differences, at which the PlotPanel objects are shown.

The list of differences contains more, since double clicking on any item in the dialogue gives more information about that specific difference. For example, the PlotPanel objects will show the differences in a dialogue like the one shown in Figure 4.11. The idea is to select either one of the plotpanels, and press the "OK" button. This will also process the result in the list of differences. For example selecting the right plotpanel will cause the arrow icon to show up in the differences list for this difference, because the user selects to import the comparison model version in the currently opened model.

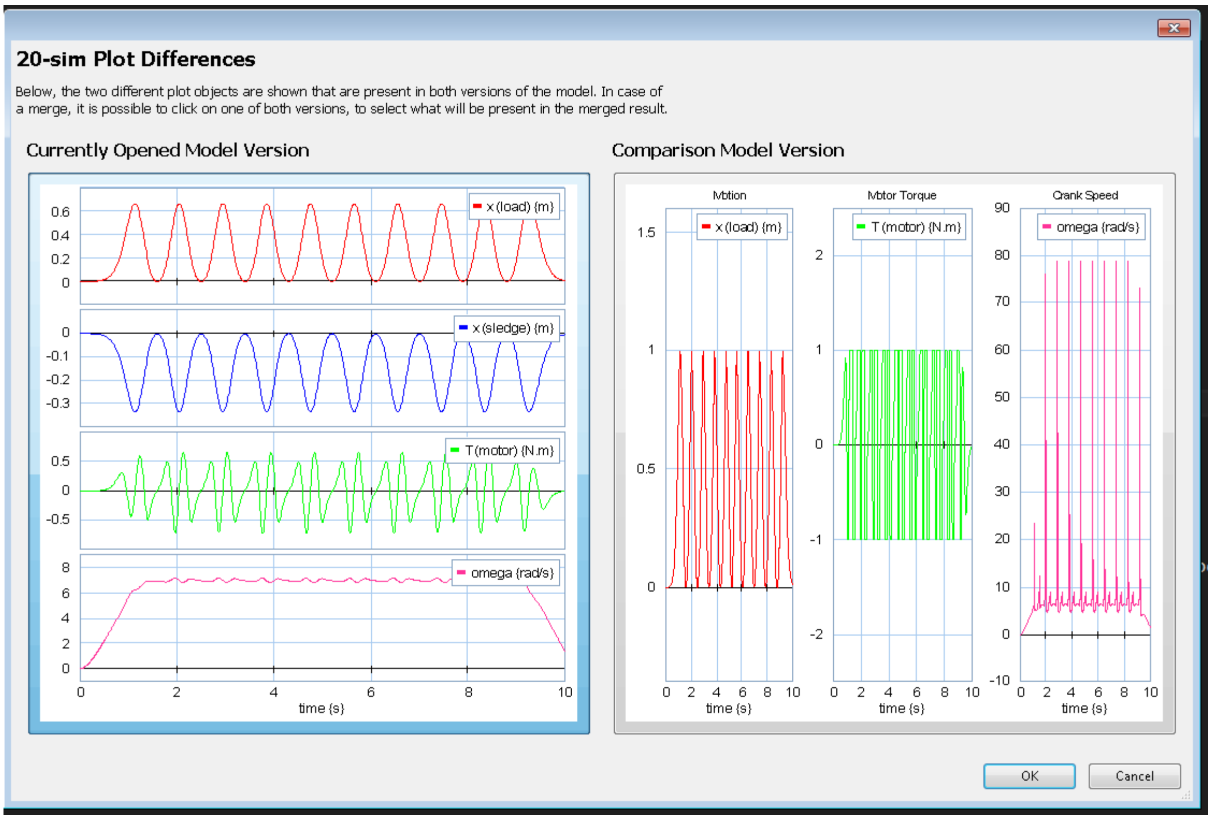
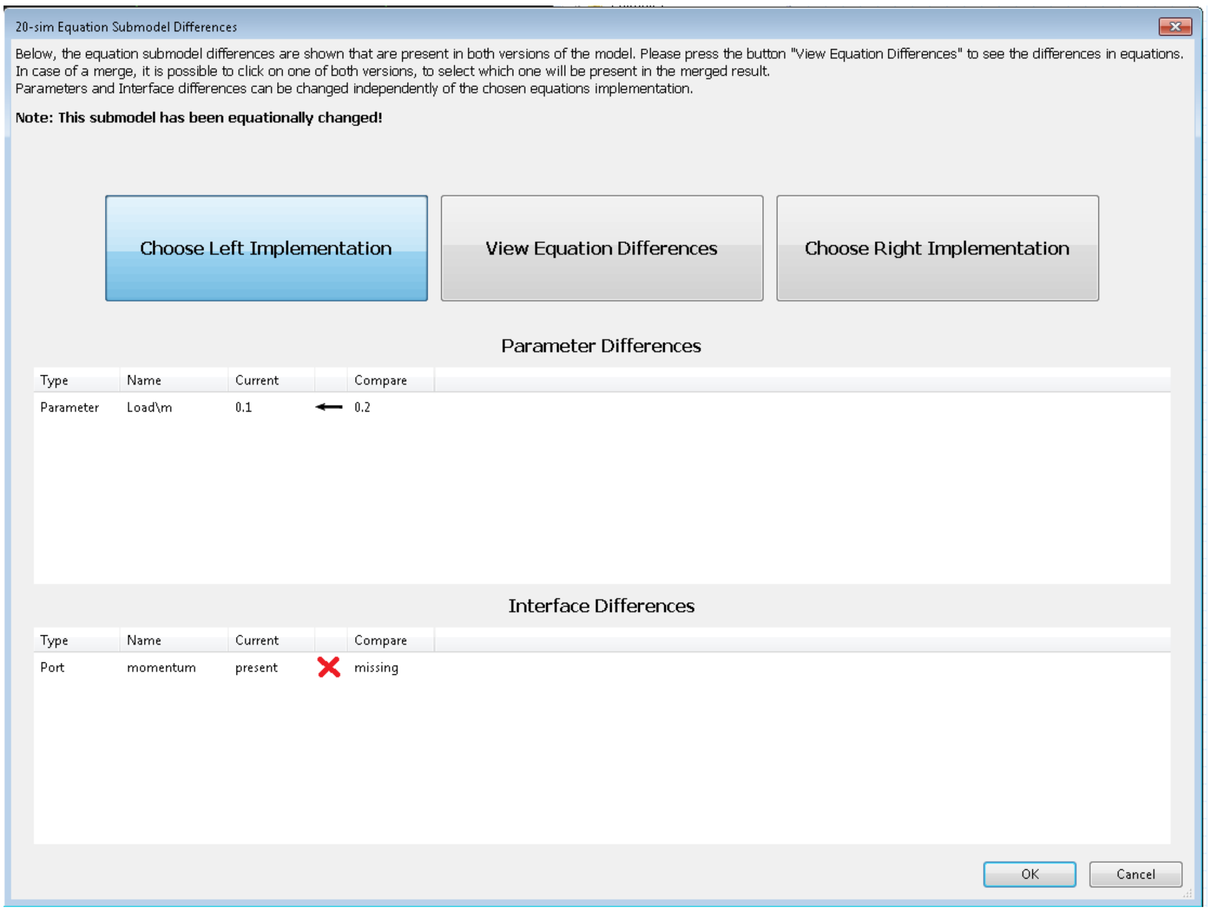

20-sim Plot Differences
Below, the two different plot objects are shown that are present in both versions of the model. In case of a merge, it is possible to click on one of both versions, to select what will be present in the merged result.
Currently Opened Model Version
Comparison Model Version
x (load) {m}
x (sledge) {m}
T(motor) {N.m}
omega {rad/s}
time {s}
Motion
Motor Torque
Crank Speed
OK
Cancel

Figure 4.11: The differences dialogue for a Plotpanel object.

For submodels, there are two types of windows: Graphical or Equational. The equation model dialogue is shown in Figure 4.12. It shows first of all two list that are similar in structure as the main differences list: an interface list and a parameters list. The parameters list indicates changed parameters, like the mass parameter for the Load submodel, which changed from 0.1 kilograms to 0.2 kilograms in the figure. The interface differences list shows ports and connections that changed between model versions. In this case, an output port called ”momentum” is present in the currently opened model, but not in the comparison model. Note that due to the fact that the computation was added for the momentum, there was an additional output relation, which caused the equational submodel to change operationally. This has been indicated by the message in bold with the text: ”Note: This submodel has been equationally changed”. Furthermore, there is the option to show the set of equations of both sides, by using the ”View Equation Differences” button. This will show the equations in the tool WinMerge (2013). One can also make a selection about which version of the equations should be in the merged result, by either selecting ”Choose Left Implementation” or ”Choose Right Implementation”. It is also possible to change parameters, ports and connections separately from the choice made for the left or right implementation. This means effectively that the Left Implementation could be chosen with the right value of the mass parameter, namely 0.2 kilograms.

Graphical submodels have their own dialogue. In these dialogues, there are also once more lists for the parameter differences and interface differences. What makes a graphical submodel dialogue unique however, is the fact that it shows the graphical representation of the changed graphical submodel. An example of what such a dialogue looks like, is shown in Figure 4.13. Note that in the case of this figure, there is a graphical model present in the currently opened 20-sim model, but there was no such graphical model in the comparison model. This means that the graphical model is removed from the model, as indicated by the ”Not present” image.

The Graphical User Interface is designed using wxWidgets. wxWidgets is an open-source and free application, with cross-platform support. Its main, unique selling point is that they use the the native platform API to create the dialogues, whereas providing a framework for typing platform independent code. More information about wxWidgets can be found in Appendix C.


20-sim Equation Submodel Differences

Below, the equation submodel differences are shown that are present in both versions of the model. Please press the button "View Equation Differences" to see the differences in equations.
In case of a merge, it is possible to click on one of both versions, to select which one will be present in the merged result.
Parameters and Interface differences can be changed independently of the chosen equations implementation.

Note: This submodel has been equationally changed!

Choose Left Implementation
View Equation Differences
Choose Right Implementation

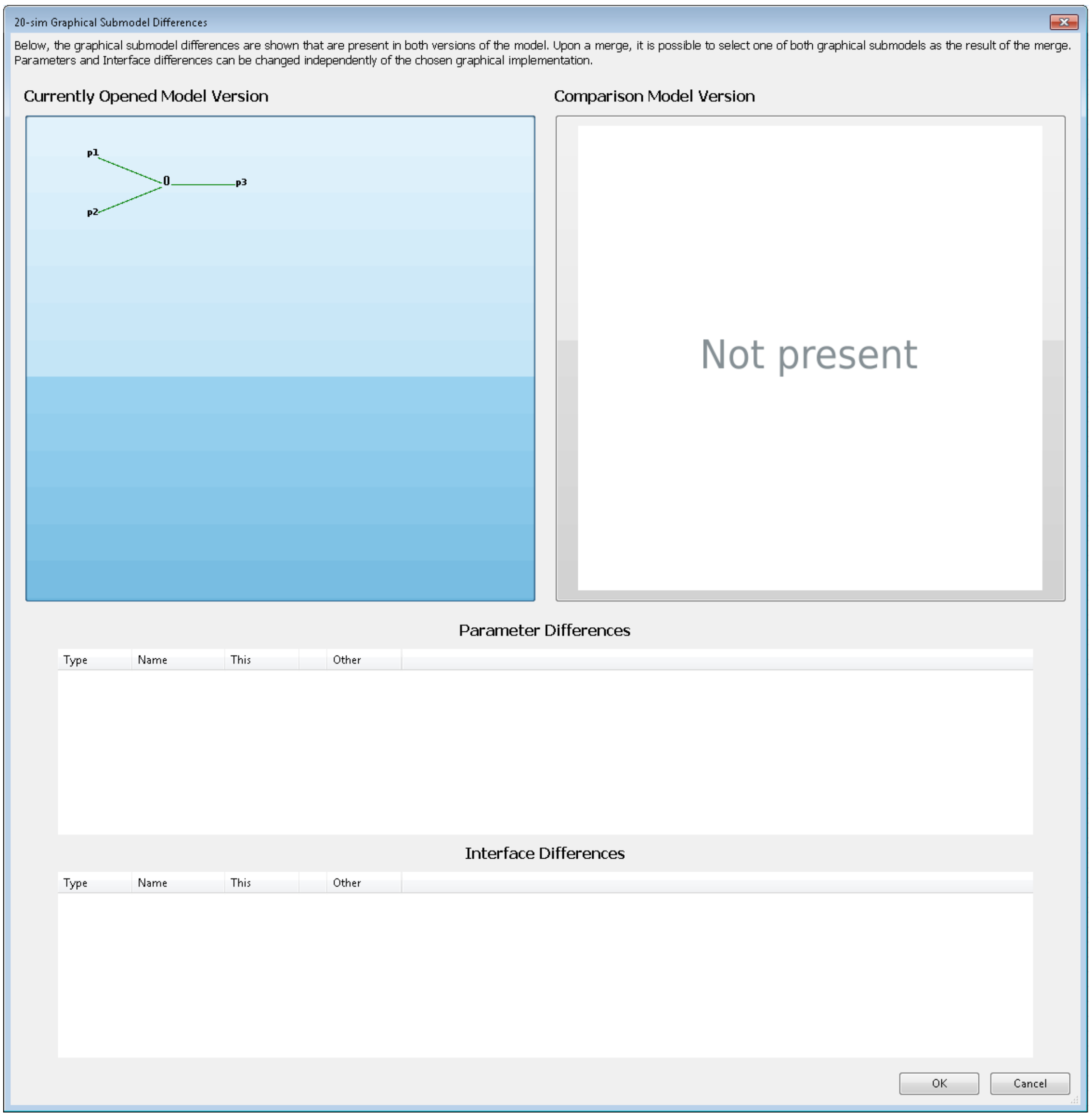
Parameter Differences

Type	Name	Current		Compare
Parameter	Load\m	0.1	←	0.2

Interface Differences

Type	Name	Current		Compare
Port	momentum	present	✗	missing

Figure 4.12: The differences dialogue for Equational submodel objects.



20-sim Graphical Submodel Differences

Below, the graphical submodel differences are shown that are present in both versions of the model. Upon a merge, it is possible to select one of both graphical submodels as the result of the merge. Parameters and Interface differences can be changed independently of the chosen graphical implementation.

Currently Opened Model Version

Comparison Model Version

Not present

Parameter Differences

Type	Name	This	Other

Interface Differences

Type	Name	This	Other

Figure 4.13: The differences dialogue for Graphical submodel objects.

Chapter 5

Results

This chapter shows the results from a set of tests that have been performed to evaluate the capabilities of the tool. The first section is about failure tests, in which the separate parts of the tool are tested by a large set of test models to see how often this part of the tool results in a failure. The second section goes into more detail on the performance of the overall tool with respect to the time it needs and the memory it allocates. Finally, the third section contains tests from a more user-perspective. These tests manually evaluate if the differences that are shown are complete and accurate, and if the impact analysis shows proper results.

5.1 Failure tests

A set of 135 models, taken from among others the examples of 20-sim and some customer projects of Controllab Products, were used for evaluating the performance of the tool. For each operation of the tool, a test is designed that shows if the models get through that part of the comparison tool. The results are indicated in a percentage that indicates how many of the models that entered that test, resulted in a pass.

5.1.1 Conversion from old to new format

The conversion from the old to the new format consists of three steps:

1. Validating that the old format (EMX) satisfies to the XSD specification.
2. Converting from the old to the new format.
3. Validating that the new format (TOMF) satisifes to the XSD specification.

These three steps were performed on all sample models, and 130 out of 135 models passed the first test, which means a score of 96.3%. The first three models failed in representing if-statements. The fourth model failed in not inserting arguments for an operator (which means that the operator does not apply to anything). These four models thus failed during the second validation step. The fifth and final model already failed in the conversion step, in which the XML generated from the grapheditor part of 20-sim could not be properly generated.

5.1.2 Reading in the XSD-XML tree

Taking out the five models that did not convert properly, 130 models are left for the second test. The second test consists out of creating the complete XSD-XML tree for each model. By doing this, one model out of 130 models failed this test, resulting in a score of 99.2%. This model failed on interpreting a sequence in a model.

One interesting aspect of this part of the comparison tool, is that this specific piece of the tool was written in a generic fashion. This was tested out by using the tool on another XSD, namely one of the image representation format SVG (Scalable Vector Graphics). SVG is also based on XML, and it has its own XSD. This SVG representation was used in the new format for storing the icons shown for submodels in 20-sim, and for storing the image representation of the contents of a submodel. This latter

Model Failure Category	Number of Models	Percentage
Failed during Conversion	5	3.7%
Failed during XSD-XML tree building	1	0.7%
Failed during Applying Context	0	0%
Failed during Operational Impact	31	23.0%
Succesfully Passed all Tests	98	72.6%
Total	135	100%

Table 5.1: Table showing at what test a specific model failed given 135 initial test models. Note that a failure in a certain row means that it first passed all tests above.

option is for example used to create the images for the graphical model differences dialogue, as shown in Figure 4.13. The XSD for this SVG format was embedded within the tool without problems, by simply importing this SVG XSD in the XSD metamodel used for the new format.

5.1.3 Applying context

Once the XSD-XML tree has been read, the next step is to apply context to it (as discussed in section 4.5). Applying context did not make any additional models from the 129 models fail, thus giving a 100% passing score for this specific step.

5.1.4 Operational impact

The operational impact problem was unfortunately the cause of more failures. Here is a summary of all the encountered failures:

- Multiple models failed, because they had unknown operators in their equations, that the comparison tool did not know of yet. These operators were added once they were encountered in this new set of models. This procedure was repeated for all models in the test set of models, such that these failures could all be resolved by simply expanding the known operators list of the comparison tool.
- 11 failures were found in reading the variables from a set of equations. These were all related to the fact that the kind (input, output, parameter, and so on) and value tags of a variable object were optional, although the code did expect at some points that these were required. This has been fixed, thus resolving all 11 failures.
- Finally, matrix and vector support in the impact analysis is so far not implemented. This is a plan for the future, but simply required too much time to be able to fit in the master assignment. This resulted in 31 failures during the impact analysis test.

There are thus 31 failures left, which were all caused by the fact that matrices and vectors were not yet implemented in the operation impact analysis. These failures have been examined manually for each of these 31 models in a debug environment, to make sure that there was no additional bug during this testing phase. All 31 models really failed purely on missing this feature in the tool. This means a score of 76% given that the test started with the 129 models that came through all previous tests. On the other hand, these are easy to resolve once the vectors and matrices will be implemented in the impact analysis in the future.

5.1.5 Failure tests overview

The overall result is summarised in Table 5.1. Overall, 72.6% of al models fed into the tool went through the tool without any problems. For purely difference testing (so without the impact analysis), the tool already goes as high as 95.6%. The large differences between these two percentages (more than 20 percentage points) merely lies in the fact that matrix and vector support for the impact analysis is not yet embedded in the tool.

5.2 Performance tests

This section describes the performance tests for fifty 20-sim models of various initial file sizes and various amount of equations. The first section describes what benchmarks are measured, and how they are measured. The second section evaluates the timing of the tool, whereas the third section evaluates the dynamically allocated process memory of the tool. Finally, a section is dedicated to analysing these results.

5.2.1 Benchmarks

Two main benchmarks are defined, which are described below. These benchmarks form the main line of identifying the performance for varying models when going through the tool designed in this master assignment.

- **Time** - The time that the tool needs to convert the old model to the new model, identify the differences, and perform the operational impact analysis is the first performance benchmark for the tool. This time is started at the point that the "Next" button in the tool is pressed, and is done as soon as all results are computed by the tool. These results include the conversion results of both models, reading in both XSD-XML trees, applying the context, and performing both the operational impact analysis as well as the impact score estimation. This time is measured within the Debugger in Visual studio 2015 in debug mode. In practice, the tool will actually be quicker, since the debug mode also takes some of the time. What is more important however, is the relative timing between models. This can be accurately monitored with this time at milliseconds precision (although the results in this report are rounded to a precision of 100 milliseconds).
- **Process Memory** - During the process, memory is dynamically allocated to hold the XSD-XML tree structure. There is also memory allocated for side-things like the graphical user interface, and using the debug mode of visual studio 2015. Once more, the relative process memory between various models can however still be determined though. The allocated memory is evaluated after showing the differences to the user. This is the point in which both XML-XSD trees and their corresponding context blocks are in this allocated memory. Another advantage of taking this point as the reference, is that after the graphical user interface shows all results to the user, a steady-state in the memory-allocation is found. Nothing is allocated or deallocated from this memory until the user performs another action. This makes the reading of the memory allocation accurate, as it is not changing over time, and thus easy to measure.

The two dependent variables against which these benchmarks are tested are the size of the original EMX model that is inserted in the tool in kilobytes, and the amount of equations present in the model. These two variables are described below.

- **Size of the EMX model** The size of the EMX model was taken as a first variable that might influence the time and memory allocation. The main reason for this choice is that the XML parsing scales directly with the amount of tags present in the EMX model. Furthermore, the more plotpanels or submodel, the more work the tool has to do. A model with a larger size is more likely to contain more of these components, and thus is also more likely to use more memory and take a longer time to find all differences.
- **Amount of equations** The amount of equations can be a dependent variable that indicates if the time and memory usage scale with this set of equations. The impact analysis for example heavily depends on this set of equations to determine if a model or submodel was operationally changed. Furthermore, equations are in fact generated twice during the conversion. The new format contains the SIDOPS code as the user types it in 20-sim, which is storing the visual representation of the SIDOPS code, and it contains the processed version of this SIDOPS code in XML form.

There are four tests, in which each combination of a dependent variable versus a benchmark is tried out. An error analysis is performed by computing a standard deviation and comparing this value for four types of fits: Linear, Logarithmic, Exponential and to the Power. The best fit is displayed as a trend line through the data points. The standard deviation is computed with the usual formula, as shown in equation (5.1).

Type of fit	Standard Deviation (size of EMX model)	Standard deviation (amount of equations)
Linear	2.06 s	1.87 s
Logarithmic	2.71 s	2.99 s
Exponential	3.46 s	2.67 s
Power	2.39 s	2.66 s

Table 5.2: Table showing the standard deviations of both plots for the timing with respect to several different types of trend lines.

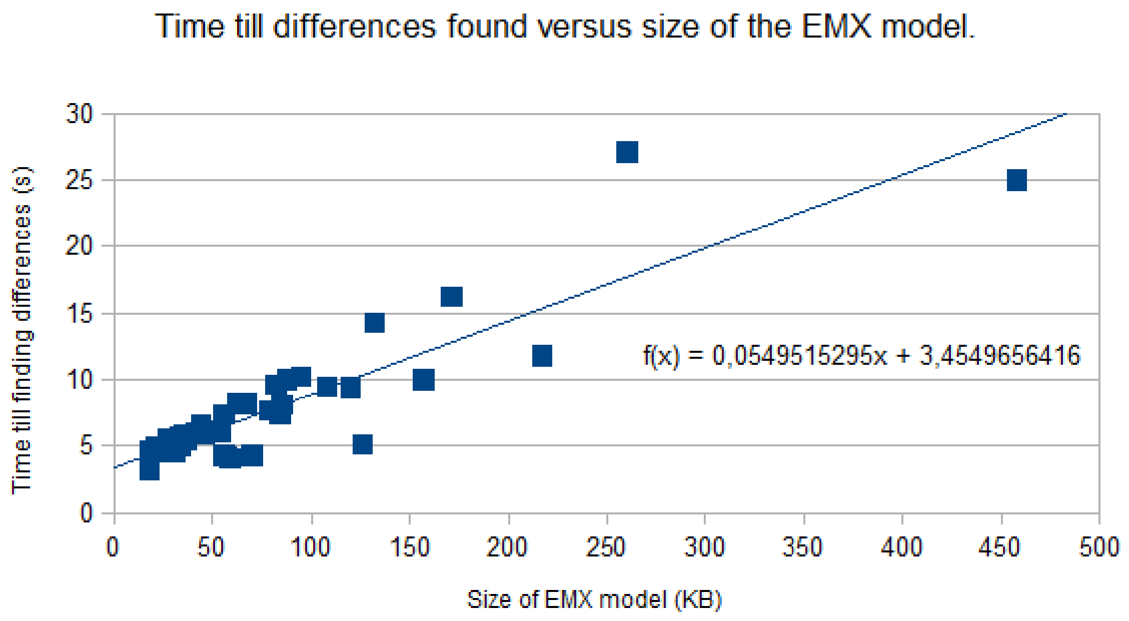


Figure 5.1: The plot of the time it takes to go through the whole process of difference finding versus the size in kilobytes of the initially inserted model. A linear trend line was found to be the best fit.

$$S_N = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2} \tag{5.1}$$

The formulas for determining the best fit were computed by LibreOffice Calc (which is an open-source variant on Microsoft Excel). The test is performed by comparing a model with itself, such that the test can be performed for one model at a time. The model itself is converted twice, and then twice loaded into memory. After that, the context is applied to both model trees. Next, each equation model performs the operational impact analysis even though the actual, final result of this impact analysis is never requested by the tool, since there are no differences between a model and itself.

5.2.2 Timing the tool

For timing the tool, all 50 models were timed and these times were plotted versus both dependent variables. The plot of the time of finding these differences versus the size of the emx model inserted in the tool is given in Figure 5.1. To get to this plot, the sample points for all 50 models were plotted, and a trend line was picked based on the standard deviations of the points with respect to the four types of trend lines. The second column of Table 5.2 shows these standard deviations for this plot for four types of trend lines. Given the fact that the Linear trend line shows the lowest standard deviation, this would be the trend line that fits the data the best.

The second plot of the time versus the amount of equations in the model is shown in Figure 5.2. Once more, as shown in column 3 of Table 5.2, the linear trend line is the best fitting one.

Overall, the time plots show an almost flat slope. The ratios of both plots are about 1 to 20. This is a good sign, since this means that expanding the model by either 20 equations or 20 KB only results

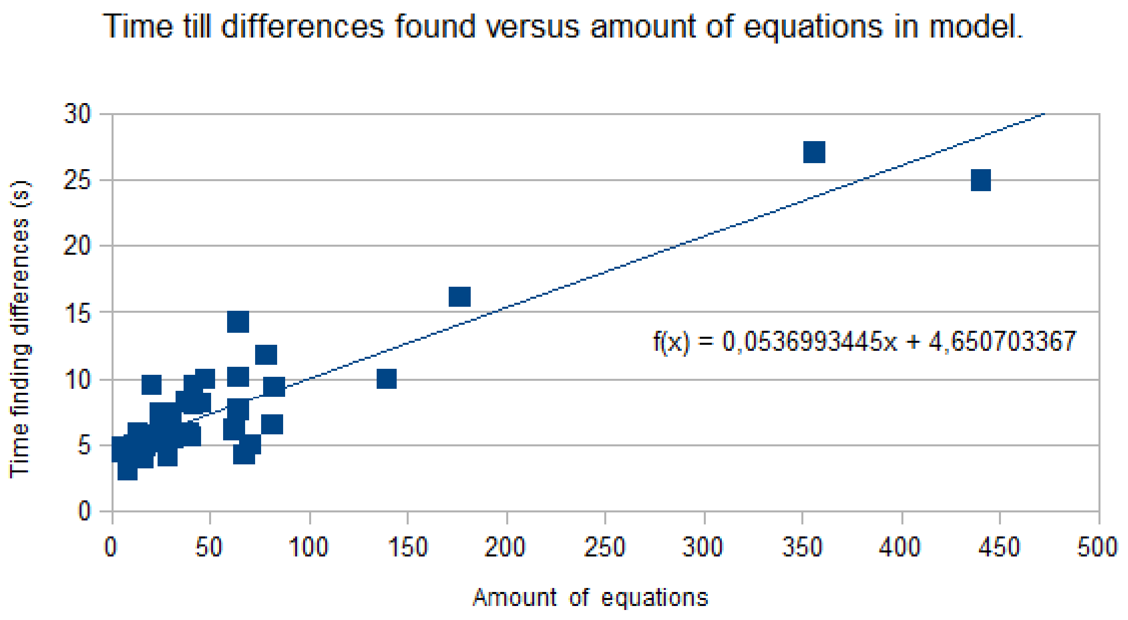


Figure 5.2: The plot of the time it takes to go through the whole process of difference finding versus the amount of equations present in the model. A linear trend line was found to be the best fit.

Type of fit	Standard Deviation (size of EMX model)	Standard deviation (amount of equations)
Linear	19.43 MB	12.47 MB
Logarithmic	33.85 MB	34.99 MB
Exponential	47.69 MB	35.37 MB
Power	26.31 MB	26.76 MB

Table 5.3: Table showing the standard deviations of both plots for the allocated memory with respect to several different types of trend lines.

in an increment of 1 second of time in the tool. Unfortunately, this result did not work for much larger models. For this assignment, one model was available that was considered a large model (1558 KB), and it took over 4 minutes to get through the tool in debug mode (compared to the expected 1 minute 26 seconds that the trend line would point out). This does indicate that there is a different behaviour for larger models, but unfortunately there was no access to more of such models during this assignment to further test this.

5.2.3 Process memory usage

Once more, all 50 models were used to determine in this case the dynamically allocated memory at the moment the user sees the generated list of differences. The result of the plot of used process memory versus the initial size of the EMX model fed into the tool is given in Figure 5.3. The standard deviations in terms of MB of dynamic memory with respect to the trend line were computed, and the results are shown in Table 5.3. Column 2 shows the standard deviations for the plot with the size of the EMX model on the X-axis, which shows that a linear trend line suits the data points the best.

The second figure with the amount of equations present in the model on the X-axis, is shown in Figure 5.4. Table 5.3 shows in column 3 the list of standard deviations for this plot. The linear fit for the trend line is also in this case the best fit for the data.

Both plots for the dynamic memory allocation have similar slopes of about 0.69-0.7. This means that an increment in size of the initial model or an increment in amount of equations is propagated through to the dynamic memory with a factor of about 0.7. For example, an increment in initial model size of 10 kilobytes would cause 7 megabyte of additional dynamic memory allocation. Unfortunately, this linear fit of the trend line also does not hold for larger models. The same model as discussed in the previous section with an initial model size of 1558 kilobytes causes about 2500 MB to be allocated (instead of the

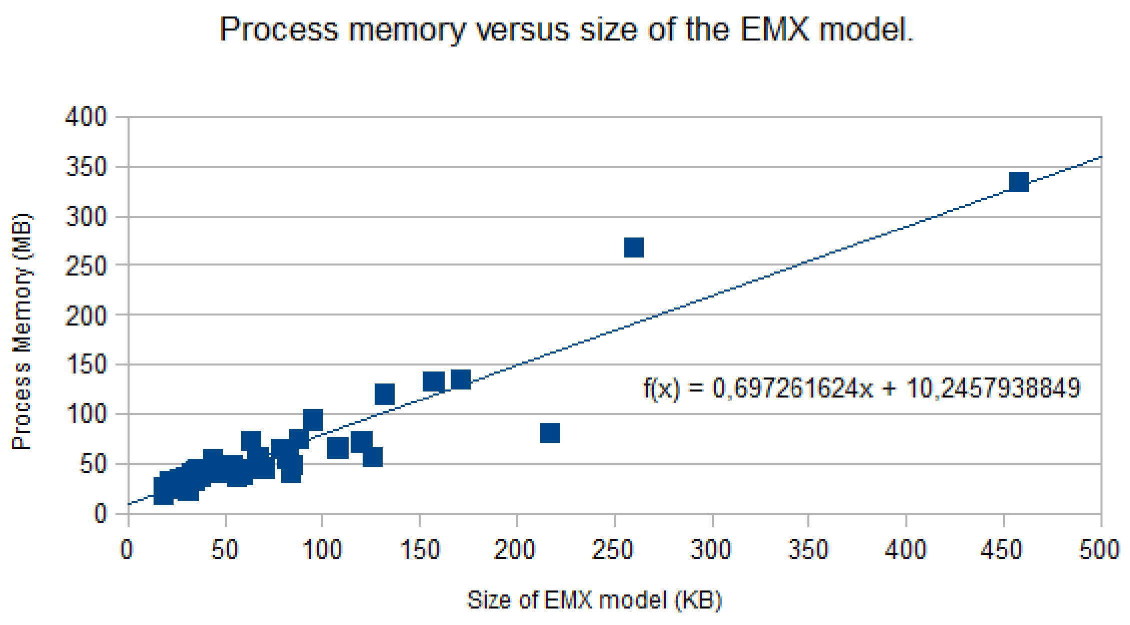


Figure 5.3: The plot of the memory allocated at the point of showing the list of differences versus the size in kilobytes of the initially inserted model. A linear trend line was found to be the best fit.

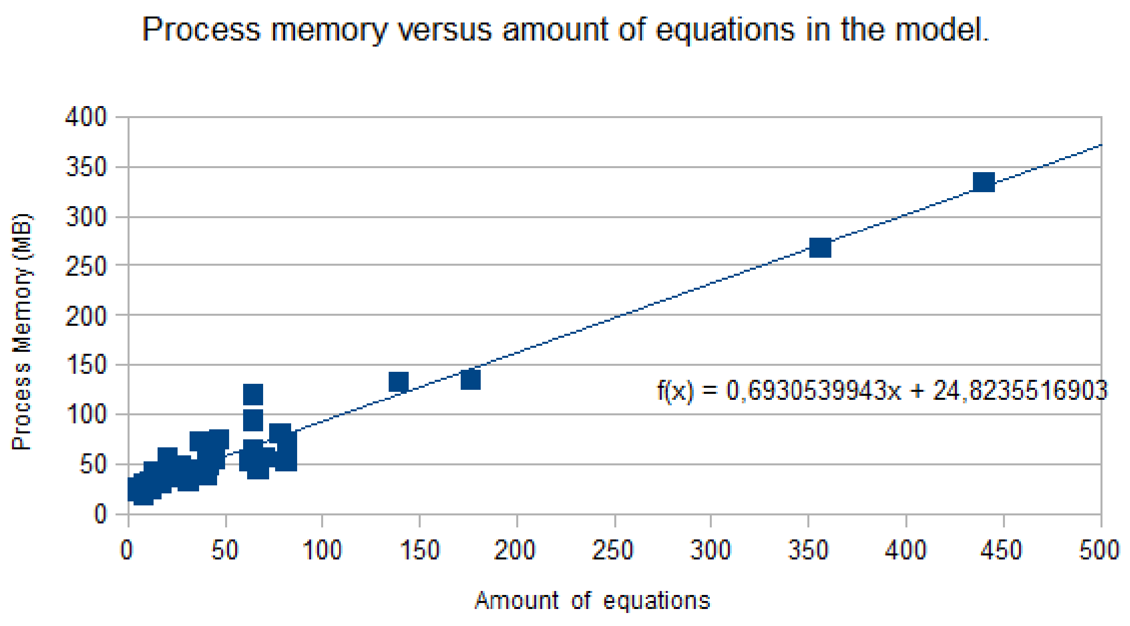


Figure 5.4: The plot of the memory allocated at the point of showing the list of differences versus the size in kilobytes of the initially inserted model. A linear trend line was found to be the best fit.

Type of difference	Amount of occurences	Amount of failures
Update Plot	16	0
Update Parameter	2	0
Add Graphical Model	1	0
Update Graphical Model	2	2
Add Equational Model	16	0
Update Equational Model	6	0
Remove Equational Model	2	0
Remove Port	4	2
Equational Submodel Equationally changed	4	0
Equational Submodel Not Equationally changed	2	0
Unchanged Submodels	11	0

Table 5.4: Table with all found differences in between versions of a model of a customer project from Controllab. The last column indicates the amount of failures for the given type of difference.

expected amount of 1097 MB).

5.2.4 Performance results overview

The standard deviations from the trend lines are smaller in both cases for the plots that used the amount of equations as the dependent variable. This suggests that the relation between the amount of equations and both the dynamic memory allocation and the timing of the tool are more linearly related to each other than when taking the original EMX file size as the dependent variable.

Furthermore, this section indicated what the expected behaviour is of the tool with respect to timing and memory usage for small models under about 200 KB of initial size or 200 equations. Even though the exact numbers are lower in practice due to the debug information from Visual Studio 2015, the relative relations still remain to exist.

5.3 Accuracy

This section evaluates the results that the tool generates. This means that it evaluates if the results are correct, and if this is not the case, what the impact is of this failure on the tools credibility. The first section is about validating the set of found differences. Are there differences missing or are there differences which are not really differences? The next section is about verifying the impact analysis: does it show false positives (it says to be operationally different, but it is not) or false negatives (it does not say to be operationally different, but it is)? Finally, there is a section about the impact score. Does it indicate an impact score that might be expected by persons that do not know how the impact score system works?

5.3.1 Verifying differences

To test if the differences that are delivered by the tool are decent, Controllab provided some models from customers, that were divided into several versions. Because of non-disclosure, these models and the customers are not published. However, in Table 5.4 an overview is given of all found differences sorted by type and transformation operation (add, update or delete). Furthermore, this table also indicates for each updated submodel if it was equationally changed according to the equations dialogue or not. Finally, the last row indicates submodels that were indicated as difference in the table, but in fact did not show any differences.

These unchanged submodel differences are often caused by adding junctions, splitters or adders in 20-sim. These have the goal to do something with all inputs or all outputs, without specifying the exact variables for the inputs and outputs. For example, the adder adds all inputs, and places the sum on the output. The equations also only show that the sum of all inputs has to be taken. Removing one input from this sum thus does not show up in the equations, however it is definitely a valid difference.

Another thing to note is that fortunately all operationally changed equation models were also labelled as such; there was no false negative. Indicating that a submodel did not operationally change, even though it actually did can give the user the wrong indication about the relevance of this difference.

The failures in removing a port (which were actually additions of a port, but shown as removals of a port), and the two updates of graphical models that failed are just faulty, and have to be fixed.

5.3.2 Verifying impact scores

Currently, there are four factors that determine the overall impact score. These four factors were already discussed in section 4.7. Because these factors are either true or not, the total amount of combinations possible with these four factors is $2^4 = 16$ combinations. This means that every submodel difference is part of one of these 16 states. Because the tool does not support adding user-based learning examples yet via the graphical user interface, the result of being in a certain state always results in the same impact score. All these currently fixed impact scores are shown in Table 5.5 for their corresponding state.

	no implementation no parameter	no implementation parameter	implementation no parameter	implementation parameter
no equation no port	0%	20%	20%	50%
no equation port	30%	60%	60%	80%
equation no port	50%	70%	70%	80%
equation port	70%	90%	90%	100%

Table 5.5: Table that indicates the given impact scores for all 16 possible states. Note that "no" here means that there was no difference of the given type, thus the first column has no implementation change and no parameter change, whereas the first row has no operational changes to the equations and no changes in the ports of that submodel.

These impact scores are based on initial guesses towards simulation impact. Each of the 16 possible states currently has 4 learning examples connected to it. These learning examples are manually chosen impact scores that would suite the impact state. Based on these learning examples, a state is inserted into the tool. The tool then computes the impact score based on the learning examples it knows. This is how the impact scores of Table 5.5 were computed.

It is already possible with the current algorithm to add any number of factors to the set of factors to determine the impact score. These factors can also have any range. The current factors were all boolean (either true or false, thus a range of values from 0 to 1), meaning they have a range of 2 numbers. However, it is possible to add for example a percentage change between the simulation results in model version 1 versus the simulation results in model version 2, and indicate it within a range of 0 to 100 percent. Furthermore, given the current four factors, it is also possible to change the outcome by adding other learning examples. Even though the graphical user interface does not support this yet, the actual code underneath has no problem with this. It is for example possible to ask several users to indicate for a set of differences how relevant they think the difference is. If these examples would be added to the set of learning examples, then a much broader view on what is a proper impact score can be given. The "Discussion" and "Conclusions and Recommendations" sections continues to try to find a better way of identifying impact depending on the actual reason for making the model.

Chapter 6

Discussion

The original main goal of this master assignment was defined as follows: ”To ease the proces of model management for a user that would like to focus on the development of models rather than the management of them.”. Even though a step has been made in obtaining this goal, there is still quite some work to do to get to a level of model management that is minimal for the user. This master assignment did provide the reader with an analysis on how to use an existing version control system as the basics of this model management trajectory. Furthermore, the comparison tool satisfies the second subgoal, that was defined as: ”giving the user a means to compare two versions of the same model”. This step has been made into a proof of concept, showing that it possible to use 20-sim and compare two versions of a model for differences.

What this master assignment however did not provide is a full solution to the two other subgoals, even though both were touched upon. The first subgoal was defined to ”give the user a means of easily storing and retrieving versions of a model”. Some scripts were made in the version control system GIT that showed that this was possible, and that all actions of a file-based version control system could be reused for a model-based version control system with the exception of model comparison. However, a full-blown implementation of model-based version control was not developed during this master assignment.

The third subgoal was defined as: ”giving the user a means to link his model to external factors as results, code, tests, and requirements”. This factor was investigated during the literature research in the form of the concepts traceability and provenance. These concepts will be discussed in line with this master assignment in the section below about literature research.

6.1 Literature research

Taentzer et al. (2014) already mentioned the concept of model transformations. These atomic operations in a tool were not present in the case of 20-sim. On the other hand, Taentzer et al. (2014) did mention the three basic operations that Brosch et al. (2010) also mentioned: add, delete and update. These operations were in the case of the article of Taentzer et al. (2014) created by using these model transformations. This master assignment however proved that finding differences is possible without the use of model transformations. This was done by looking at the two model files, creating a model tree out of them with the help of a metamodel, and then comparing those trees for differences.

Reiter et al. (2007) was one of the only literature papers that referred to the concept of impact. His context was to see what the impact of syntactical changes were on the semantics of a model. This impact analysis became quite a large part of this master assignment, as it was something completely new for commercial model comparison tools. Even though the power of impact was not clear from the start of the master assignment, it opened possibilities that were not possible to expect beforehand. The most important unexpected result here was the fact that by simply monitoring if a submodel operationally changed, it was possible to identify which external results were outdated and had to be regenerated. In the light of model management, a tool that can monitor for you when simulation results, generated code or unit tests change their behaviour is one of the most promising results of this master assignment.The actual implementation of traceability in the context of model management however was not performed during this master assignment.

Provenance was defined by Shamdasani et al. (2014) in three stages: tracking changes to provide historical records of actions performed, the outcomes achieved, and design decisions taken. The first step has been the most important one throughout this master assignment. 20-sim can now record actions performed between two model versions, and can use this as the basics for providing historical records in an existing file-based version control system. The second step is quite closely related to traceability, in which the outcomes achieved are coupled to the changing model. Finally, the third step of monitoring the design decisions was not yet discussed during this master assignment. Adding a small message to storing a new version of a model already helps to identify later on why the change was made in the first place. This can for example make the difference between the idea of a sandbox mode (introduced in section 3.1) versus a production mode. The sandbox mode was originally for the purpose of quickly testing something out in a model. One could however extend this idea by using sandbox mode as some initial model building without adding these provenance messages, and use another mode called production mode for the actual official development in which messages can be added to a new model version about the reason of development of that specific version.

6.2 Design phase

One of the things that could have been done different was the fact that all aspects of this master assignment (as discussed in the Implementation chapter) were each developed independently. At the end, it became more difficult to combine the separate aspects of the tool together into one working overall model comparison tool. One of the main reasons for this might already lie in the design phase, because of the choice of designing the graphical user interface in the last part of my master assignment. The graphical user interface often is a good way of thinking about what actions a user could take, and how they should relate to the code underneath. Even though it was mostly clear where the master assignment would head towards (even up to a phase in which impact would be added), it might have been more convenient to think about this main line of user interaction and cooperation of the different tool components.

Another discussion that is specific to the Design phase lies in the fact that there is always the tradeoff between context and complexity. From a user perspective, the tool should be easy to comprehend. Context needs to be applied to obtain user-intuitive concepts like a submodel or plot. On the other hand, putting everything under context is a lot of work. One has to find those specific contextual blocks that are important and intuitive to the user, without holding him back in functionality of comparing models. A good example are curves that are plotted in a plot in 20-sim. A curve can change its line colour, the font of the text of the curve in the legend, thickness and much more. On the one side, giving the user the option to see these differences and merge them is a good thing, on the other hand the question should be posed if these differences are relevant enough to show. Other important changes like changing the variable that is plotted are much more relevant to show. One has to ask if it is a problem that the colour of a curve cannot be set in the tool, when it comes at the price of focussing the user his attention to the really relevant differences between model versions.

There is another tradeoff worth mentioning: the tradeoff between building a separate tool for finding differences versus embedding the results of the model comparison tool in the application itself. The idea is that colour indications in 20-sim could already show which submodels have changed. Furthermore, equations can be shown per version once an equation submodel is selected in 20-sim. These equations can then be compared within 20-sim in a file-based manner. By using the context of 20-sim, it is possible to show every difference exactly at the position that it has changed within the user-familiar environment of 20-sim. A change in the properties dialogue of a submodel can actually be shown exactly at that dialogue opened in 20-sim. The major drawback of this approach however is that the user can change things in the model that alter the results that are shown in the model comparison results. This means that the comparison results get outdated once the user alters something in the model. To be able to resolve this, one could make 20-sim read-only during a comparison analysis or constantly update the differences list once users update the model.

6.3 Results

The results are promising in the sense that they show that a model comparison tool can be made for 20-sim that still achieves good performance, while being able to support most types of models. One should make the sidenote that the tool has not been tested much for very large models. This is something that might have improved the results chapter, since then more tests of industrial practice could be used to verify the workings of the tool. Furthermore, it was difficult to obtain decent results for the abstract concept of impact, since there is not an exact definition of what impact means to a user. The main difficulty here lies in the fact that not one model is perfectly competent to model all behaviour that can be observed. A model has a problem context in which it is accurately modelling the physical behaviour. As long as this context is unknown to the tool, determining an impact score is difficult. For example, a user that wants to perform a hardware in the loop simulation is interested in other aspects of the model than someone performing a simulation to find the natural frequencies of a physical system.

Interestingly, due to the linear fit of data through the performance curves, it can be seen that for EMX models under about 250 kilobyte or 200 equations, quite a good performance can be achieved. First of all, the fact that a linear fit can be made is already a good result at itself. There is no quadratic increment in waiting time for the tool or dynamic memory usage when the size of the EMX model or the amount of equations increases. One should however note that adding 100 kilobytes of initial EMX model size only increments the time to wait by about 5 seconds and the process memory by a about 70 MB. The same results also apply to an increment of 100 equations.

Chapter 7

Conclusions and Recommendations

7.1 Conclusions

This master assignment showed several aspects of model management. First of all, an initial design of model management within an existing version control system was developed, which actually showed that the desired project structure as described in section 3.3 could be achieved within GIT. The subgoal about model version comparison (subgoal 2), that was defined in section 1.1, was succesfully expanded into a full-blown proof of concept that actually did exactly that: comparing model versions. Furthermore, concepts like traceability and provenance were used to find initial ideas on how to use the results of model management to improve the modelling workflow itself, by identifying traces from a model to its related external dependencies like requirements, tests and simulation results.

The new format for a 20-sim model, that was specifically designed for this model comparison tool, is also a format that is very useful for 20-sim. Not only is part of the old EMX format that was still plain text converted to an XML representation, also the format itself is more transparent. The format is more separated into functional components in 20-sim, and the combination of the XSD and the model file itself form the complete set of information that 20-sim needs to know to open, process and simulate the model. The XSD model remains the same for all models, and contains information about the default value of tags that are missing in the actual model file. A proper conversion was also made during this master assignment from the old to the new format.

The separate components of the merge tool each were designed in such a way that they were also useful as separate tool. The XSD-XML tree builder application for example also works on random other XSD files. This was already shown in section 5.1.2, in which the XSD of an image storage type called SVG was embedded within the tool without problems. This behaviour can however also be achieved on any random XSD and XML, given the fact that they do not use unsupported components in their XSD. Also, adding the context layer seemed to be a good approach on reproducing the behaviour of model transformations: it becomes possible to identify components of a 20-sim model, and say something about if they were added, removed or altered.

The impact analysis proved to be a new feature for model comparison tools, in the sense that there are no competitor model comparison tools that actually say anything about the impact of a difference on the model behaviour. Even though the ideas posed in the impact aspect are still initial ideas on how to approach this problem, the consequences of giving feedback to the user about impact and being able to trace outdated results are a major step towards easier model management. Furthermore, computing an impact score was based on a technique from artificial intelligence, which means that the user can even change the outcome of the impact score per difference type, based on his own preferences.

Finally, a graphical user interface was made that more concrete shows the workflow that a user takes: deciding which two models to compare, deciding if a merge of these two models should be performed, observing the differences, and deciding on further actions about what to do with the comparison results.

7.2 Recommendations

One of the most interesting aspects to do future research on is the impact analysis of a difference between two model versions. By improving this aspect of the tool, it becomes possible to determine not only if a difference is more relevant than another, but also to determine if results generated by a model are still up-to-date or might be no longer fitting the model or submodel that they were originally created from. The idea that a difference can be detected that automatically lays these new traces to new versions of the results of a model makes for a very powerful version of model management, especially when these results are automatically regenerated by the tool itself.

Another topic that is useful to investigate, is how requirements engineering fits into this picture. The concept of traceability was frequently used to explain that there might be so-called traces in between components of a 20-sim model and the results produced by those components. Traceability is however the full path from requirements to acceptance tests. This report mainly focused at the consequences of a change in a 20-sim component on the results of that component, but the component in 20-sim is a result of a requirement for that component in the first place. Placing this master assignment in a larger context by including the full path from the requirements of the model to the acceptance tests that verify that the 20-sim model satisfies these requirements makes the complete traceability path complete.

Furthermore, 20-sim does not support model transformations. Monitoring these atomic actions would make the process of finding differences a lot easier. The solution proposed in this master assignment had to work around this problem by using the comparison of two models via their metamodel, and applying context to identify user-intuitive 20-sim components like submodels and ports. In the best case scenario, a set of model transformations between two model versions is the exact transformation from one model version to the other. The discussion already touched upon embedding the results of the differences analysis within 20-sim itself by for example colour indicators that show which submodels have changed. The problem there was a synchronisation between users altering something in the tool, and the fact that this outdates the results shown by the comparison tool. This synchronisation however would be a lot easier if model transformations could be monitored. By monitoring these model transformations, it becomes possible to do a specific synchronisation action based on what the user altered in the model.

Based on the discussion, already some different modes of model-development were mentioned. Among others the difference between a sandbox and production mode. The production mode is the mode in which formal development is done, in which it is important to describe the reason of altering the model. On the other hand, the sandbox mode is there to do some testing or trial and error without being bothered with writing the reason for the model development. Other modes of model-development are closely related to the purpose of model development, and determining a decent impact score. Some models are there purely to analyse the physical behaviour of a system, whereas others were created to perform a hardware in the loop simulation. Even other models can end up as trainingsimulators, in which the main goal is to teach others the workings of the system. These all have different ideas of impact. In a physical simulation, the main goal is to analyse the response of the physical plant, and thus changes in simulation are important. A hardware in the loop simulation however is more focussed on using the model as part of a larger set of programs. The behaviour of this controller is important, but testing out the limits of the model and the safety behaviour of the physical system are in this case just as important. By defining several modes (physical simulation, hardware in the loop, trainingsimulator, and so on), it might become easier to determine an impact score of certain differences between model versions.

As a last recommendation, the vision on the tool for the future is described. In the complete context of model management there should also be aspects of the tool that focus on the management of versions. The ideal tool for model management would be a simple dialogue with a slider that goes through the versions over time of that model. The ideal goal would be to take any version anywhere in time by using this slider, and then be able to work further from that specific model version. Furthermore, specific models that are more important than others should be able to be tagged in this window by simply selecting a model version and tagging it. The switch between sandbox mode and regular mode (or production mode) should be a simple checkbox in this dialogue or maybe even a button or dropdown menu in the 20-sim toolbar itself. Finally, the fact that collaboration often requires a shared model structure is something that can be embedded in this dialogue too. It is for example possible to see the whole linear history of the currently opened model either from a local perspective (only showing the model predecessors that were stored on the computer itself) or remote perspective (showing the model predecessors that are stored locally and the predecessors that were made by others and taken from a remote model structure).

Appendix A

Batch Scripts Version Control

This section will shortly describe the commands needed to do model-driven version control with the command-line tool of GIT. This section will not cover complete samples, but rather the reasoning behind the made choices in the scripts. Note that this section expects the reader to have some basic notion of the commands in GIT. Chacon and Straub (2014) give a complete manual about everything related to GIT, including the introductory commands.

A.1 Setting up the project structure

The idea of a project structure, as was described in section 3.3, will be the starting point of this section. First of all, GIT has to create a repository, and there has to be something in there to be able to switch to another branch. The following listing shows these commands:

```
git init
echo 'test' > ProjectConfig.xml
git add ProjectConfig.xml
git commit -m '[master] Add project configuration file'
```

This creates an empty GIT repository, and adds a ProjectConfig.xml file with the contents "test". Furthermore, the file is staged and committed to the GIT repository. The -m flag indicates that the commit message is specified directly afterwards. The next step is to make a branch for the first model, called test_model.emx in this case:

```
git checkout -b test_model
git rm -r *
echo '' > test_model.emx
git add test_model.emx
git commit -m '[test_model] Added test_model.emx'
```

First, checkout to the branch called test_model. Note that the -b flag automatically creates a new branch with that name, before going there. Remove everything from the branch (it is a copy of the master branch, and the ProjectConfig.xml file is not needed here), and then stage the test_model.emx file. Finally, commit it to the test_model branch. Now, it is time to create the master_models branch, and add the test_model branch as a tree underneath it:

```
git checkout -b master_models
git rm -r *
git commit -m 'Emptied master_models branch'
git read-tree --prefix= -u test_model
git commit -m '[master_models] Add the test_model to master_models.'
```

First, checkout to a new branch called master_models branch. This branch will monitor all separate 20-sim models, and will function also as the Models folder as indicated in Figure 3.3. Remove all contents in this branch, and commit the empty branch. Next, the actual subtree merging action is done. This action was already described in section 4.1, and it copies the tree of a specific branch and pastes it somewhere in the tree of another branch. The command read-tree exactly does this. The –prefix flag indicates the folder in the master_models branch in which the resulting test_model tree should be pasted.

In this case this is simply in the root directory, thus this option is left empty. The -u flag is used to indicate the branch tree that should be copied. Finally, the results have to be committed. Note that staging is not necessary, since in fact the read-tree command is the low-level version of the staging command.

A similar action has to be performed to couple the master_models branch under the master branch:

```
git checkout master
git read-tree --prefix=models/ -u master_models
git commit -m '[master]␣Commit␣models␣to␣project␣folder␣models'
```

Go to the master branch, call the read-tree command again. This time however, the models folder has to be made, in which the master_models branch contents have to be stored. Finally, do another commit. At this point, the exact project structure as shown in Figure 3.3 has been reproduced in GIT.

A.2 Adding a new model

To add a new model to the project structure, first again create a new branch (in this case called new_model) and check it out:

```
git checkout -b new_model
git rm -r *
echo '' > new_model.emx
git add new_model.emx
git commit -m '[new_model]␣Added␣new_model.emx'
```

Add the new model, stage it and commit it to the new_model branch. Once more, subtree merging has to be done to add the new model to the master_models branch:

```
git checkout master_models
git read-tree --prefix= -u new_model
git commit -m '[master_models]␣Added␣new_model.emx␣to␣master_models␣branch'
```

Checkout to the master_models branch, and once more use subtree merging to add the new_model branch tree to the root folder of the master_models branch. Also, commit these results. Now, the next step is different, since this time no tree has to be added to the models folder, it has to be merged. The reason for this, is that there already exists a tree in the folder models, and to be able to add the new results to that tree, both trees have to be merged. This is done via the git merge command:

```
git checkout master
git merge --squash -s recursive -Xsubtree=models master_models
git commit -m '[master]␣Merged␣master_models␣into␣the␣models␣folder.'
```

The –squash flag indicates pulling in external changes and adding a precommit message for these changes. The -s recursive indicates to use the recursive merge strategy (although this one will likely be overkill, since only model files are present, no folders). The -Xsubtree flag indicates in what folder the merge is about to happen, in this case the models folder. Next, the branch to be merged is specified to be the master_models branch. Finally, commit the results. At this point, the new model is added to the project structure.

A.3 Add a new version of an existing model

Besides adding new models, it is often desired to add a new version of an already existing model. This is quite similar to updating a model. The first step is again going to the specific model branch.

```
git checkout test_model
echo 'test' > test_model.emx
git add test_model.emx
git commit -m '[test_model]␣Changed␣test_model.emx'
```

The model is updated, and again staged and committed. This is how usually a new version is added to GIT. To propagate the results upwards all the way to the master branch, the master_models branch has to be updated as well:

```
git checkout master_models
git merge --squash -s recursive test_model
git commit -m '[master_models] Merged test_model to master_models branch'
```

Note that a merge has to be used again, since the master_models branch already contained a test_model, and thus the new version has to be merged with the old version. Finally, the master branch has to be updated again:

```
git checkout master
git merge --squash -s recursive -Xsubtree=models master_models
git commit -m '[master] Merged test_model to master_models branch'
```

Note that one important comment has to be made here: A merge will result in a conflict. This means that one has to take into account that there are two merges here: from the test_model to the master_models branch and from the master_models branch to the master branch. The merge has to be done twice. The merge should result in two identical models. The solution to this challenge is not part of this master assignment.

A.4 Show two versions of a model

This section will discuss the challenge of showing two versions of a model next to each other, whereas GIT cannot checkout two versions of the same branch. This problem has been solved as follows:

```
if exists: rm -r diff
git checkout test_model
git log (remember first 7 characters of previous commit)
git checkout master_models
git read-tree --prefix=diff/old/ -u f769f34
start "" "notepad++.exe" diff/old/test_model.emx
git read-tree --prefix=diff/new/ -u test_model
start "" "notepad++.exe" diff/new/test_model.emx
```

This listing is more pseudo-code. Given the fact that the current location is the master branch, remove the folder diff, that is used for storing temporary difference results (if this script was already executed before, the diff folder has to be removed). This folder will be recreated later on in this script. Go to the test_model branch and use the log command to find a commit that is to be compared. Remember the first 7 characters of its commit hash, and checkout to the master_models branch. Use the read-tree command to place the old commit (indicated by the remembered 7 characters hash) in the diff/old folder. Then start notepad (or notepad ++ in this case) with the path to the old version of the model. Next, do the same action for the most recent version of the test_model model, and place it in the diff/new folder. Also start this file in notepad. Both versions can now be viewed at the same time. Of course, these files can just as easily be opened in 20-sim next to each other in two instances of 20-sim when they contain actual models.

Appendix B

XSD files and Comparing Models

An xml schema design file, or for short xsd file, is a file that contains a complete definition for a set of xml files. XSD files have quite some constructs that can be benificial to describing the specification for a set XML files, however the tool described in this master assignment only makes use of five:

- **Element** - The main construct in an XSD, which represents a one to one relation with the XML tag. The XML can give additional information about this tag, for example its minimum and maximum amount of occurences in the XML or its default value.
- **Attribute** - The construct in an XSD to indicate that a tag in XML has an attribute.
- **Sequence** - A construct indicating that the child XML tags should be present in the specific order given.
- **Choice** - A construct indicating that one (and only one) of the childs of this choice should be present in the XML.
- **Group** - A comfort construct, in which certain sets of XSD constructs that occur often, can be labelled with a group name and easily placed at many spots throughout the XSD. If the group is updated, it will automatically change at all places it is referred to in the XSD.

To make this into an application that can read in an XSD tree with corresponding XML in an automated fashion, these five XSD constructs were changed into C++ classes. The idea is to recursively build up the tree as it was created in the XSD, and along the way read in the corresponding XML tags from the XML file. Because of the fact that such an XSD component should be able to call the same functions on all of its (also XSD) child components, it is useful to use inheritance here, and build one base-class for all types of XSD components. In that way, every component class contains at least a limited set of shared functions for all XSD components.

The C++ application stores quite some information, but at this point there are three functions that are especially relevant for this section:

- The constructor of a component. This is the point where a new link in the XSD tree can be created. The constructor makes sure that all relevant information is stored about the component under creation. Things like the minimum and maximum amount of occurences, the tag name, the tag content, if it has any children, what the parent component is and so on are all present in every component that needs them.
- **FindNextNodes** is the function that searches for the next nodes, in fact the child nodes of the current node. In Figure 4.2 for example, the child node of the Parameters tag, is the left sequence.
- **FindSiblings** is the function that searches for sibling nodes. As was seen in Figure 4.2, there are situations in which a component can occur more than once. This is then indicated by a number under the component. The sequence in this figure can for example occur more than once. If this sequence occurs more than once, it should be found by the application. The resulting XML on the right side of this figure shows an example of siblings, in which the Parameter XML tag and its children Name and Value occur three times. The first found Parameter tag thus has two more siblings in the XML.

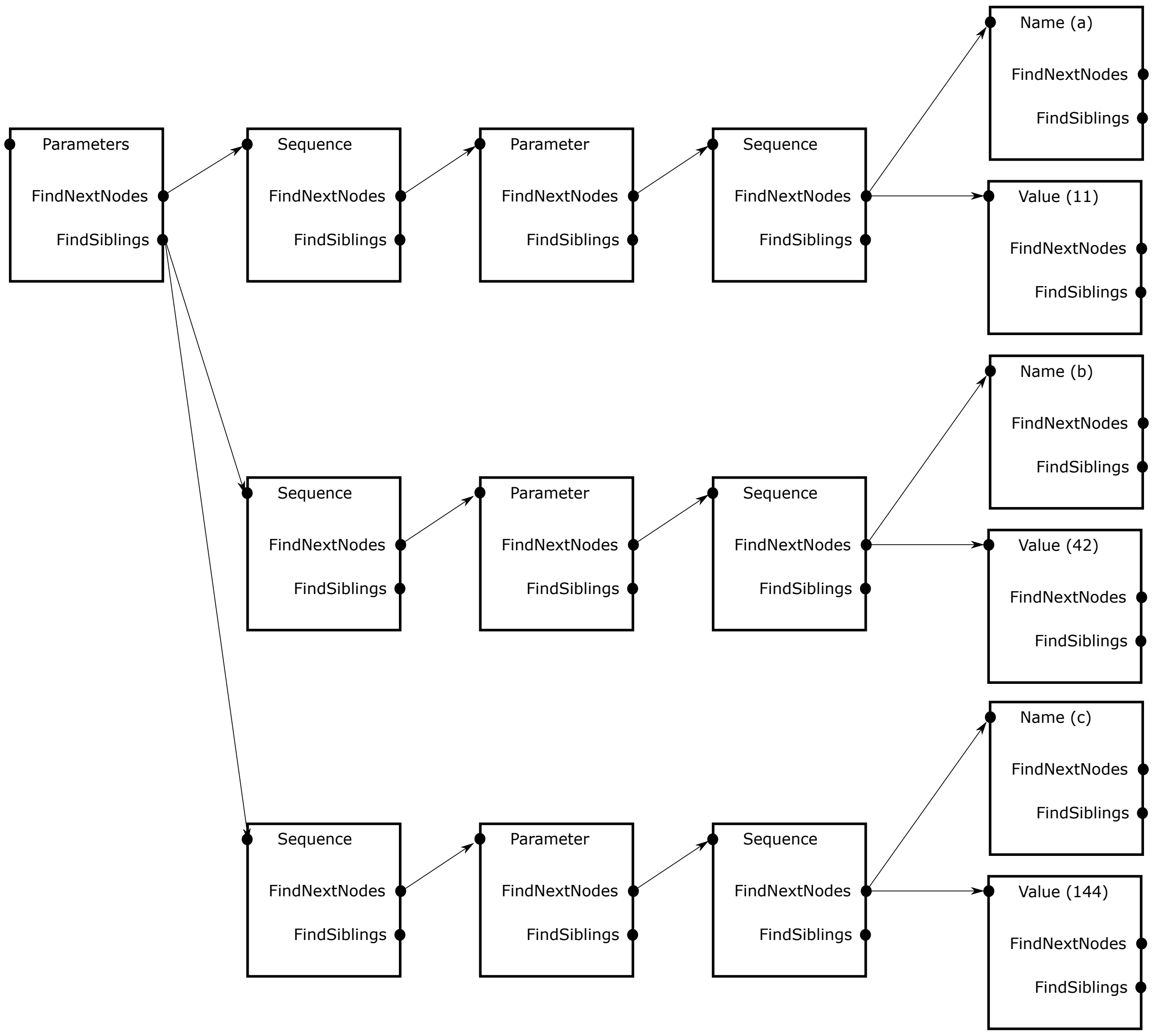


Figure B.1: A diagram on how the XML in Figure 4.2 is reconstructed in a data structure in C++. For the Name and Value tags, the corresponding values are given between brackets. The dots indicate function calls. Dots on the left are incoming function calls from other classes, dots on the right are outgoing function calls to new child or sibling classes.

Take for example Figure 4.2. This figure can be used as an example to indicate what function calls are made and in what order, as can be seen in Figure B.1. Note that the left sequence is the component to have the siblings, not the tag Parameter as the XML might suggest. This is due to the XSD specifying the multiplicity of the sequence to be ranging from 0 to infinity, whereas the Parameter tag can occur only once. Making more siblings for the sequence, gives the multiple instances of Parameter in the XML.

Appendix C

External Tooling

This appendix is dedicated to an explanation about the external tools that were used for completing my master assignment, as well as their purpose in the comparison tool development.

C.1 20-sim

20-sim (2016) is a simulation program for physical systems. It is targeted at the engineer with minor programming knowledge. The user-interface is oriented towards simplicity, while still being able to do powerful simulations with several options and toolboxes. The main simulation routine is based on a port-based approach, in which the computation routine uses energy flows and proper causality assignments to come up with the optimal set of equations that corresponds to the physical model made. Another feature is that every submodel block, even the ones from the library, can be altered at equation level. This allows the user to change any block into the behaviour desired for the model under development.

20-sim is the program at which the comparison tool in this master assignment is applied to. 20-sim was produced at Controllab Products B.V., at which I did my master assignment. The combination of scientific research and having access to the internals of 20-sim, made it possible to realise a difference and merge tool that is both satisfying the modern standards of model merging according to the literature, as well as applicable to an actually commercially available modelling tool, namely 20-sim. This combination of using state-of-the-art literature research with the possibility to alter the program itself forms a powerful combination in which it actually becomes possible to apply all literature to a practical application.

C.2 TinyXml2

TinyXml2 (2015) is a parser for xml files. With its minimal source code, it is a quick alternative to the larger programs for xml parsing. TinyXml2 was made as a second version to the popular xml-parser TinyXml. TinyXml had quite some teething trouble, but in TinyXml2 they used those problems to make a new xml-parser that accounted for these problems.

TinyXml2 was used as the main foundation in my master assignment when it comes to xml parsing. The emx-converter application made use of this parser to parse the EMX format, and to write to the new format. Furthermore, TinyXml2 was used in the xsd-merge tool, in which both the xsd, as well as both input xml files (either of the old or new 20-sim format) were parsed using TinyXml2, and the merged file in the new format was also written once more using TinyXml2. Finally, XML-parsing was used to get a set of generated equations from an xml-file generated by the simulation tool of 20-sim (for more information about 20-sim, see the previous section).

C.3 xmllint

Xmllint (2015) is a tool that is part of the libxml2 library. In fact, its original purpose is to parse an xml-document, but its functionality is too limited to actually manipulate those xml files. The reason that this tool was used however, is that it is a command-line application for validating an xml versus the XSD metamodel file. An XSD contains information about what should be present in an xml document, in what order, and how often is a tag allowed to be present in the xml. More information about xml schema design files, can be found in Appendix B.

During the scope of my master assignment, there were many stages that needed an xml to be validated against an xsd. The first is when an emx file is converted into the new format. Before the conversion, there is a need to validate the emx format, and after the conversion there is a need to validate the newly generated file in the new format. Moreover, there is also a need for validation in the actual merging tool. Both input files in the new format should be valid, as well as the file that results from the merge.

C.4 wxWidgets

wxWidgets (2016) is a set of functions that offer users a framework for working with multiple platforms (mostly windows, linux, and OS X), while only using one set of functions. wxWidgets advertises with this framework, and indicates that they lay the focus on building cross-platform graphical user interfaces. This is also why this master assignment made use of wxWidgets, because it is an open-source and free-to-use graphical user interface builder.

C.5 GIT

GIT is a version control system based on a distributed approach. Distributed technically means that every user has its own copy of a piece of software, in this report a model, and they can each change it and send so-called patches (a set of changes) to other users such that they can apply it to their own local version. In practice, distributed version control systems like GIT also support the option to make a remote server that stores a global copy on which all users work together to achieve the final result. The local usage in the sense that each user can have a local offline back-up is though still present in those remote cases.

The choice for GIT over other version control systems was made based on its speed, large user community, and its unique branching model. A branch in GIT is a complete copy of everything under version control that can be separately altered with respect to any other branch present. The overhead for a branch in GIT is minimal (it takes 40 bytes, which only resembles a 40 character long hash stored as a reference to the branch created). This little overhead means that branching is a much more common action in GIT than it is in the other version control systems, which each make an actual physical copy of all data. This branching strategy allows for making a branch, just to try something out, and then either delete the changes if they do not appeal, or merge them back to the master branch for further use. For more information about what was exactly done with GIT in this master assignment, take a look at Appendix A.

Appendix D

Bayesian Learning

Usually, probability theory works with given data samples. These data samples are then complete enough to determine a final probability. The theory in this chapter however is concerned with computing probabilities of which there are missing variables, called latent variables. The theory of this chapter originates from the field of biology. Medical experiments for example have limited test information on patients available to draw a conclusion on. Even though little literature is available about applying this theorem on software applications, one will soon see that determining an impact score for a difference between two models is quite similar.

This chapter will first discuss some concept definitions. Then, Bayes' formula will be explained, which determines the conditional probability of an assertion given a set of data examples. The final step in this chapter will be to go into more depth on how to approximate the probability given by Bayes' formula using the expectation-maximization algorithm.

D.1 Concepts

The technique described in this document is a technique for parameter estimation, given a set of learning examples. The parameter to be estimated is called the target feature; all parameters on which this target feature depends that are known beforehand are called known parameters. A parameter that is unknown at the time that the target feature should be estimated, is called a latent parameter. The symbols are taken from Do and Batzoglou (2008), as well as the concepts of a latent parameter and observed or known parameter. The term target feature is taken from Yang (1997).

Take m the amount of experiments performed, and s the amount of known parameters for a certain experiment, then the known (or observed) parameters input vector can be described as:

$$\vec{x} = \{x_{i,j} | 1 < i < m, 1 < j < s\} \tag{D.1}$$

The latent parameters can be formally described as well, given p being the amount of latent parameters:

$$\vec{z} = \{z_{i,k} | 1 < i < m, 1 < k < p\} \tag{D.2}$$

Finally, the target feature values can also be caught in a vector, with q the number of target features, and t the number of values a target feature can have:

$$\vec{\theta} = \{\theta_{l,c} | 1 < l < q, 1 < c < t\} \tag{D.3}$$

Example D.1.1: Biased coins
Say that there are two coins with different biases, such that flipping each coin gives different probabilities for throwing heads or tails. Now take the result of the flip as known parameter (i.e. this is what is known), and the coin with which is thrown as the latent variable (i.e. this is not known). Finally, take as a target feature the probability that a certain coin results in heads. This means that the vector of known parameters, the vector of latent parameters, and the vector of target features could for example be:

$$\vec{x} = \begin{bmatrix} heads \\ tails \\ heads \\ heads \\ tails \end{bmatrix}, \vec{z} = \begin{bmatrix} coinA \\ coinA \\ coinB \\ coinA \\ coinB \end{bmatrix}, \text{ and } \vec{\theta} = \begin{bmatrix} \frac{2}{3} \\ \frac{1}{2} \end{bmatrix}$$

D.2 Bayes' formula

Bayes formula is a way of computing a probability that some hypothesis is true, given a set of examples with information about this hypothesis. Before continuing, it is important that all target features are mutually exclusive:

$$\vec{\theta} = \{\theta_{1,1}, \cdots, \theta_{q,s} | \theta_{i_1,j_1} \cap \theta_{i_2,j_2} \text{ for } (i_1, j_1) \neq (i_2, j_2)\} \tag{D.4}$$

Bayes formula, as given by (Yang (1997)) and (Osoba et al. (2011)), now states the following (in which $\vec{e}$ is a set of experiments to learn the probabilities from):

$$P(\theta_{i,j}|\vec{e}) = \frac{P(\vec{e}|\theta_{i,j}) \cdot P(\theta_{i,j})}{P(\vec{e})} \tag{D.5}$$

In which $P(\vec{e}|\theta_{i,j})$ is the probability that, given that the target feature has a certain value, the set of examples is exactly reproduced. $P(\theta_{i,j})$ is the set of initial probabilities for the target features, and $P(\vec{e})$ the probability to reproduce the given set of examples. The resulting probability, $P(\theta_{i,j}|\vec{e})$, is the probability that, given a set of learning examples, the target feature $\theta_{i,j}$ has the desired value.

D.3 Expectation-maximization

The expectation-maximization algorithm is a way of finding a lower bound for the probability $P(\vec{e}|\theta_{i,j})$, and then maximizing this lower-bound function to come-up with a new value for $\theta_{i,j}$. Expectation-maximization is an iterative algorithm, which means that it has to be run in iterations, until the final value for the target feature value probabilities becomes stable. This means that the two-step approach of finding a lower bound, and maximizing the resulting function to come up with a new $\theta_{i,j}$ is done in every iteration. Do and Batzoglou (2008) explain how the expectation maximization algorithm works, which is where the rest of this chapter is based on.

First, a distribution is chosen that suits the problem at hand: the Dirichlet distribution. The Dirichlet distribution was chosen based on chapter 7.8 of the book of Poole and Mackworth (2010), in which a similar problem in Bayesian learning was performed succesfully, using also the Dirichlet distribution. The Dirichlet distribution is very closely related to a binomial distribution, and is characterised by a set of probabilities, which is the set of probabilities $\vec{\theta}$, and a set of counts, which is the set of counts $\vec{n}$ as described below (q is the set of target features, and t the set of possible values for target feature value l):

$$\vec{n} = \{n_{0,0}, \cdots, n_{l,c} | 1 < l < q, 1 < c < t\} \tag{D.6}$$

Note that the size of the probability vector equation (D.3) should be equal to the size of the counter vector equation (D.6). The Dirichlet distribution is now given by:

$$P(\vec{e}|\theta_{i,j}) = \frac{1}{B(\vec{\alpha})} \prod_{i=1}^{q} \theta_{i,j}^{\alpha_{i,j}-1} \tag{D.7}$$

In which $\vec{\alpha}$ is by definition:

$$\vec{\alpha} = \{\alpha_{0,0}, \cdots, \alpha_{q,s} | 1 < l < q, 1 < c < t, \alpha_{i,j} = n_{i,j} + 1\} \tag{D.8}$$

and in which $B(\vec{\alpha})$ is simply a constant to normalise the probabilities, which is for the rest of this chapter not relevant, for reasons about to be explained. The reason that the official definition of the Dirichlet distribution uses $\vec{\alpha}$, which is simply adding 1 to each $n_{i,j}$, is that the expectation formula can be simplified. Using $\vec{\alpha}$ has no further benefits for the rest of this chapter, and thus is also avoided for now.

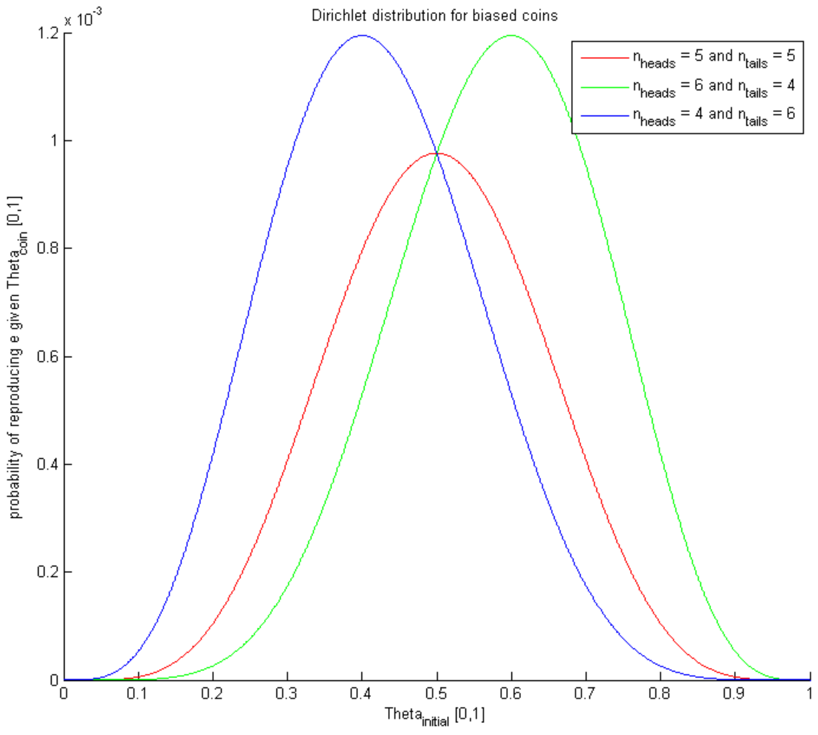


Figure D.1: The Dirichlet distribution for two dependencies (heads and tails), given different outcomes of the sample experiments.

Example D.3.1: Biased coin
Take a coin with which in three throws twice heads and once tails were obtained. The Dirichlet distribution (or in fact the Beta distribution, which is the binary version of the Dirichlet distribution) can be given by:

$$P(e|\theta) = \theta^2 \cdot (1 - \theta)^1$$

In which θ is the chance of throwing heads. The variable θ is now the variable for which to solve, since this is the target feature (the probability of throwing heads given this biased coin). Figure D.1 shows the dirichlet distributions given different outcomes of the 10 throws (as indicated by the legend in the top right corner). The x-axis is the variating initial probability that is inserted in the dirichlet distribution formula. Note that the graphs are symmetric around the initial probability of 0.5. This is due to the fact that the amount of factors is binary (either heads or tails). This means that the chance at getting n heads is exactly opposite to the chance at getting n tails given a certain initial probability. If you have a high chance at throwing heads given an initial probability, then the chance of throwing tails given that same initial probability is low.

Given this distribution, Bayes' formula is now dependent on $\theta_{i,j}$. This means that the value of $\theta_{i,j}$ is the value that is needed for the probability of the target feature. One approximation to compute Bayes' formula, is the expectation-maximization algorithm, in which $\theta_{i,j}$ is thus computed iteratively. A first step in this, is to compute some initial probabilities for $\theta_{i,j}$ for all values of i. These initial probabilities can for example be determined by doing a set of experiments and determining how often the result satisfies a certain target feature to be true. This is exactly the vector $\vec{n}$ that was defined in equation (D.6).

The expectation-maximization algorithm is based on maximising the right-hand side of Bayes' formula, and then determining the corresponding argument (i.e. $\vec{\theta}$). This means that any scaling values in Bayes' formula are irrelevant, since even though they do change the scaling of the probability function, they do not change the position of the maximum, and thus they do not change the corresponding argument. This means that $B(\vec{\alpha})$ can be ignored, since it does not depend on $\vec{\theta}$, and thus can be seen as a scaling factor. Furthermore $P(\vec{e})$ is also not dependent on $\vec{\theta}$, and thus also this term can be seen as a scaling factor. The argmax approximation can thus be written as:

$$\hat{\vec{\theta}} = \underset{\vec{\theta}}{\operatorname{argmax}} \left(P(\vec{e}|\vec{\theta}) \cdot P(\vec{\theta}) \right) \tag{D.9}$$

In which $\hat{\vec{\theta}}$ is the argument that was discussed before, which is the result of the argmax function at the next iteration of the expectation-maximization algorithm. The first step of the expectation step of the expectation-maximization algorithm is to determine the fraction that given a random sample, the probability $\theta_{i,j}$ is true for all values of i. The next step is to compensate for the given example. For example: If an example can be in three possible states, and the probability of being in state 1 is defined as $\theta_{i,1}$, and the experiment gave as a result to get twice into state 1, then the virtual amount of times that state 1 was reached for this example is $2 \cdot \theta_{i,1}$.

Example D.3.2: Biased coins

Given two biased coins with initial probabilities for throwing heads being: $\hat{\theta}_A = 0.60$, and $\hat{\theta}_B = 0.50$, and a set of throws with 5 times heads and 5 times tails, what is the probability that this set was thrown using coin A and what was the probability that this set was thrown using coin B?

Given $P(e|\theta_A) = \theta_A^5 \cdot (1 - \theta_A)^5$, being $P(e|\theta_A) \approx 7.96 \cdot 10^{-4}$ for the initial value of 0.6, and $P(e|\theta_B) = \theta_B^5 \cdot (1 - \theta_B)^5$, being $P(e|\theta_B) \approx 9.77 \cdot 10^{-4}$ for the initial value of 0.5, the fractions can be determined as follows:

$P(\theta_A|e) = \frac{7.96 \cdot 10^{-4}}{7.96 \cdot 10^{-4} + 9.77 \cdot 10^{-4}} \approx 0.45$

$P(\theta_B|e) = \frac{9.77 \cdot 10^{-4}}{7.96 \cdot 10^{-4} + 9.77 \cdot 10^{-4}} \approx 0.55$

This means that the likeliness that this throw was done using coin A is 0.45 and the likeliness that it was thrown using coin B is 0.55. Compensating this for the example, can be done by the fact that there are 10 throws out of which 5 throws were heads and 5 tails. This means that virtually $0.45 \cdot 5 \approx 2.2$ throws were resulting in throwing coin A and landing on heads, the same for tails (there were equal amounts of heads and tails) and for coin B, this means that virtually $0.55 \cdot 5 \approx 2.8$ times heads was thrown using coin B (also again the same for the amount of tails thrown with coin B).

Summing all the virtual possibilities for $\theta_{i,j}$ and a certain state is the next step to do. Determining the fraction of virtual states that satisfies $\theta_{i,j} = true$ and dividing them by all states for both $\theta_{i,j} = true$ and $\theta_{i,j} = false$ results in the new value for $\hat{\theta}_{i,j}$.

Example D.3.3: Biased coins

Given the previous example, coin A resulted in virtually 2.2 coins landing on heads, and 2.2 coins landing on tails. The new value for $\hat{\theta}_A$ for the next iteration can be determined by: $\hat{\theta}_A = \frac{2.2}{2.2+2.2} = 0.5$. The same holds for $\hat{\theta}_B = \frac{2.8}{2.8+2.8} = 0.5$. Using more learning examples than the 1 used in this example, will result in a better approximation of these values. Note that the initial value of $\hat{\theta}_A$ has changed, and thus at least one other iteration has to be performed.

D.4 The algorithm

This section will go into more depth on how the formulas described earlier in this chapter result in a meaningful output for an impact score for the merging tool. This section focusses on submodel differences between two models, and then specifically on four boolean criteria: is the submodel functionally changed, were ports added or deleted to or from the submodel, were implementations added or deleted, and does the submodel have many connections to other submodels. The latent variable here is the impact score, which ranges from 0 to 10 (in fact, this will be interpreted as 0% to 100% in the graphical user interface). The factor θ with target features is thus already containing 11 rows (for each possible value of the impact score). However, the amount of possible outcomes is not just binary anymore (i.e. heads or tails) but can be one out of sixteen possibilities (from false,false,false,false to true,true,true,true for the known parameters). The target features matrix thus becomes:

$$\vec{\theta} = \begin{bmatrix} \theta_{0,A} & \theta_{0,B} & \cdots \theta_{0,P} & \\ \theta_{1,A} & \theta_{1,B} & \cdots \theta_{1,P} & \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{10,A} & \theta_{10,B} & \cdots \theta_{10,P} & \end{bmatrix} \tag{D.10}$$

In which the first index indicates the impact score (from 0 to 11) and the second index indicates the state: A=(false,false,false,false), B=(false,false,false,true), ..., P=(true,true,true,true). The same thing can be done for the counter matrix:

$$\vec{n} = \begin{bmatrix} n_{0,A} & n_{0,B} & \cdots n_{0,P} & \\ n_{1,A} & n_{1,B} & \cdots n_{1,P} & \\ \vdots & \vdots & \ddots & \vdots \\ n_{10,A} & n_{10,B} & \cdots n_{10,P} & \end{bmatrix} \tag{D.11}$$

The algorithm now follows the following steps:

1. Find the set of initial probabilities for θ, based on learning examples (examples of which the resulting target feature value is given).
2. Determine the counts based on learning examples.
3. Determine the product of the Dirichlet distribution: $P(\vec{e}|\theta_{i,j}) = \prod_{j=1}^{q} \theta_{i,j}^{n_{i,j}}$ for all i (i.e. all values for the impact score).
4. Determine the probability fraction by dividing each column by the sum of all its column elements. This effectively normalises each column, thus giving a distribution of probability given a certain state assigned to a column.
5. Multiply the resulting matrix elementwise with the counts matrix to obtain the virtual amount of times that for a certain row i (i.e. a certain impact score) the state j (from A to P) occurs.
6. Divide each element in row i by the sum of all elements in row i. The result is the new set of $\theta_{i,j}$ probabilities.
7. Iterate through bullet point 3 till 6 until the probabilities converge or a maximum amount of iterations has been reached.

Example D.4.1: Algorithm Example

Take the following learning examples:

Experiment #	State	Impact score
1	M	6
2	B	10
3	G	6
4	A	2
5	E	3
6	B	7
7	G	7
8	B	10
9	B	7

Because this is a simplified example, take only those states (A,B,E,G,M in that order, each assigned to a column) and impact scores (2, 3, 6, 7, 10 in that order, each assigned to a row) that are relevant to the above set of learning examples. A rough estimation for the probabilities matrix can be obtained by for a certain impact state dividing the count for a certain state by the sum of all counts for that impact state:

$$\hat{\vec{\theta}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & \frac{2}{3} & 0 & \frac{1}{3} & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{D.12}$$

Also determine the matrix with counts:

$$n = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 2 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 & 0 \end{bmatrix} \tag{D.13}$$

Determine the values for the Dirichlet distribution, given the initial values of $\hat{\vec{\theta}}$:

$$P(\vec{e}|\vec{\theta}) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{4} & 0 & 0 \\ 0 & 0 & 0 & \frac{4}{27} & \frac{4}{9} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{D.14}$$

In which the element at row i and column j is computed using row i of the probabilities matrix and row j of the counter matrix. For example, row 3 column 4 is computed using the probabilities $\theta_{6,j} = \begin{bmatrix} 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$ and counts $n_{7,j} = \begin{bmatrix} 0 & 2 & 0 & 1 & 0 \end{bmatrix}$, which gives the probability: $P(\vec{e}|\vec{\theta_{i,j}}) = 0^0 \cdot 0^2 \cdot 0^0 \cdot \frac{1}{2}^1 \cdot \frac{1}{2}^0 = 0$

Then compute the fractions by dividing the columns by the sum of their elements:

$$P(\vec{e}|\vec{\theta})_{norm} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & \frac{4}{13} \\ 0 & 0 & 0 & 0 & \frac{9}{13} \end{bmatrix} \tag{D.15}$$

The next step is to compute the virtual pieces of states A,B,E,G, and M, based on the count matrix, and the normalised probabilities matrix. Note that a column of the normalised probabilities matrix coincides with the probabilities for the impact value given a certain set of counts. For example, take the fifth row of the counter matrix, which is $n_{10,j} = \begin{bmatrix} 0 & 2 & 0 & 0 & 0 \end{bmatrix}$. Then the fifth column of the normalised probability matrix indicates how this 2 times state B is divided amongst the several impact values: $\frac{4}{13}$ of 2B goes to impact value 7, and $\frac{9}{13}$ of 2B goes to impact value 10. Taking this all together, the following table can be made:

impact:	2	3	6	7	10
test # 1	1·A	0	0	0	0
test # 2	0	1·E	0	0	0
test # 3	0	0	1·G + 1·M	0	0
test # 4	0	0	0	2·B + 1·G	0
test # 5	0	0	0	$\frac{8}{13}$·B	$\frac{18}{13}$·B
total:	1·A	1·E	1·G + 1·M	$2\frac{8}{13}$·B + 1·G	$\frac{18}{13}$·B

Given these results, it is then time to round up this iteration, and compute the new values for $\theta_{i,j}$. This is done by first inserting the numbers from the table above in a matrix (note the impact values are assigned to a row each, whereas the states are again the columns):

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 2\frac{8}{13} & 0 & 1 & 0 \\ 0 & \frac{18}{13} & 0 & 0 & 0 \end{bmatrix} \tag{D.16}$$

And then finally dividing this matrix by the sum of its row:

$$\vec{\hat{\theta}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & \frac{34}{47} & 0 & \frac{13}{47} & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{D.17}$$

Note that the probabilities for the impact value of 7 (row 4) did change, and the others did not change. This was due to the fact that impact value 7 shared two states, namely state B with impact value 10, and state G with impact value 6. Impact value 10 is only related to state B according to the set of learning examples, thus it does not matter what test set you insert in the application, it can only get an impact value of 10 if the input state is B, and otherwise can never reach an impact value of 10. Due to the final normalisation step, this will always result in a probability of 1 for state B, and 0 for all other states. The independence of experiment 3 (1 times G and 1 times M) from experiment 4 (2 times B and 1 times G) is the reason that impact value 6 does not change. Experiment 3 is namely solely given to impact value 6, whereas the part resulting in state G of experiment 4 is fully given to impact value 7. The only reason that the impact value 7 row does change, is because of its share of state B in experiment 5.

Appendix E

Build Instructions

This section will show how to build the code for this master assignment. Before continuing with this section, there is one important remark to be made. The code to be build does not work with any consumer version of 20-sim yet. To obtain the proper build of 20-sim, please contact Controllab and ask for the specific 20-sim 4.7 Tom version with build number 6998. The code to be built is a Visual Studio 2015 (2016) project. Open the .sln file (the solution file). In the solution explorer window, use the right-mouse button on any project and go to the Properties dialogue. Make sure that for "Configuration" you use "All Configurations". Go to the C/C++ tab on the left of the properties dialogue, and add all paths to all the projects in this solution under "Additional Include Directories". The dependencies needed for each project are shown in Table E.1. Besides the dependencies of these projects, the GUI also needs wxWidgets 3.0.2 to operate (it might be possible to use another version of wxWidgets 3, but this has not been tested, also versions of wxWidgets 2 are not working). Add the following paths to at least the GUI projects Additional Include Directories: "path\wxWidgets\3.0.2\lib\vc_x64_lib\mswu" and "path\wxWidgets\3.0.2\include". The solution can both be built for 32-bit or 64-bit platforms.

Also note that in Visual Studio 2015, these references have to also be set as build dependency via the References option. Go to a project in the solution explorer and open it. Right-click on References, and select Add Reference. Select the above dependencies for each project by checking the correct boxes, and press OK to confirm.

E.1 Project meaning

Each project in this solution has its own task. All projects are briefly discussed to see what functionality they contain. The tinyxml2 project contains a limited set of functionality needed to use the tinyxml-2 library, and it contains the deepCopy function needed to perform a complete copy of a tinyxml-2 XML tree and paste it to another tree. HelperFunctions mainly contains some macros and Null-pointer check functions that are useful throughout most of the projects in this solution. emxConverter contains the conversion algorithm from the EMX format to the new tomf format. For this part, the specific 20-sim 4.7 Tom build with build number 6998 is needed to obtain the XML for the grapheditor from 20-sim. xsdMerge consists out of two aspects: the generic xsd-xml comparison tool, with all classes that are inheriting from the Component parent class, and the application of context, with all classes that are inheriting from the Block parent class. The Impact project contains the algorithms for the Impact analysis. The XMLStackInterpreter class is the class needed for finding operational changes, whereas the HypothesisCollector class is the class needed for determining an impact score. There is an ImpactTests class that shows some examples on how to use these classes. Application is the project that handles all communication between the user interface classes and the other fundamental classes. GUI and TUI are the projects for the user interface. The GUI class is to build the actual GUI using wxWidgets, the TUI class is used to make command line tools. The TUI class currently shown was for example used for generating test results in section 5.1.

E.2 Running the project

Once the project is built, the project can be run either in the debugger or via the just built executable. Make sure to set the start up project to either the GUI or TUI project if the executable should be

Project:	Dependencies:
Application	HelperFunctions tinyxml2 emxConverter Impact xsdMerge
GUI	Application HelperFunctions tinyxml2 emxConverter Impact xsdMerge
emxConverter	HelperFunctions tinyxml2
HelperFunctions	tinyxml2
Impact	HelperFunctions tinyxml2
tinyxml2	none
TUI	Application HelperFunctions tinyxml2 emxConverter Impact xsdMerge
xsdMerge	HelperFunctions tinyxml2

Table E.1: All dependencies between the several projects in the Visual Studio 2015 project for the comparison tool.

ran from within Visual Studio 2015 (right click on either of these projects and select "Set as startup project"). Within Visual Studio, both a debug build or release build can be run. The GUI and TUI projects need to have a command line argument, which is the model on which the comparison tool is called. This can be set within Visual Studio by going to the properties dialogue, then the debugging tab and then in the box after "Command Arguments".

References

20-sim, 2016. URL `http://www.20sim.com/`.

P. Brosch, G. Kappel, M. Seidl, K. Wieland, Wimmer M., H. Kargl, and P. Langer. Adaptable model versioning in action. *Modelliering 2010 24.26 März 2010, Klagenfurt*, 2010.

Scott Chacon and Ben Straub. *Pro Git, Second Edition.* New York: Springer, 2014. ISBN 978-1-4842-0076-6.

James Cheney, Peter Buneman, and Bertram Ludäscher. Report on the principles of provenance workshop. *ACM SIGMOD Record*, 37(1):62–65, 2008. doi: 10.1145/1374780.1374798.

Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilator. *Version Control with Subversion.* 2011. URL `http://svnbook.red-bean.com/en/1.7/svn-book.pdf`.

cplusplus.com. std::stack, 2016. URL `http://www.cplusplus.com/reference/stack/stack/`.

Mariangiola Dezani-Ciancaglini, Ross Horne, and Vladimiro Sassone. Tracing where and who provenance in linked data: A calculus. *Theoretical Computer Science*, 464:113–129, 2012. doi: 10.1016/j.tcs.2012.06.020.

Chuong B Do and Serafim Batzoglou. What is the expectation maximization algorithm? *Nature Biotechnology*, 26(8):897–899, 2008. doi: 10.1038/nbt1406.

EnSoft Corporation. Simdiff 4 team, 2005. URL `http://www.ensoftcorp.com/simdiff/`.

Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering.* London Berlin Heidelberg, 2005.

IEEE std 610.12-1990. Ieee standard glossary of software engineering terminology. page 82, 2002.

Mark McIlroy. Reverse polish notation, 2016. URL `http://mathworld.wolfram.com/ReversePolishNotation.html`.

modeling-languages.com. Version control tools for modeling artifacts, 2010. URL `http://modeling-languages.com/version-control-tools-modeling-artifacts/`.

Osonde Osoba, Sanya Mitaim, and Bart Kosko. Bayesian inference with adaptive fuzzy priors and likelihoods. *IEEE Transactions on systems, man and cybernetics - Part B: Cybernetics*, 41(5):1183–1197, 2011. doi: 10.1109/TSMCB.2011.2114879.

Brian O'Sullivan. *Mercurial: The Definitive Guide.* 2009. URL `http://hgbook.red-bean.com/read/`.

David Poole and Alan Mackworth. *Artificial Intelligence, Foundations of Computational Agents.* Cambridge University Press, 2010.

Thomas Reiter, Kerstin Altmanninger, Alexander Bergmayr, Wieland Schwinger, and Gabriele Kotsis. Models in conflict - towards a semantically enhanced version control system for models. *Lecture Notes in Computer Science*, 5002:293–304, 2007. doi: 10.1007/978-3-540-69073-3_31.

Hannes Schwarz, Jürgen Ebert, and Andreas Winter. Graph-based traceability: a comprehensive approach. *Software and Systems Modeling*, 9(4):473–492, 2010. doi: 10.1007/s10270-009-0141-4.

Jetendr Shamdasani, Andrew Branson, and Richard McClatchey. Towards provenance and traceability in cristal for hep. *Journal of Physics: Conference Series*, 513(3):679–711, 2014.

Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software & Systems Modeling*, 13(1):239–272, 2014. doi: 10.1007/s10270-012-0248-x.

TinyXml2, 2015. URL `http://www.grinninglizard.com/tinyxml2/`.

Visual Studio 2015, 2016. URL `https://www.visualstudio.com`.

Sven Wenzel. Unique identification of elements in evolving software models. *Software & Systems Modeling*, 13(2):679–711, 2014. doi: 10.1007/s10270-012-0311-7.

Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, 9(4):529–565, 2010. doi: 10.1007/s10270-009-0145-0.

WinMerge, 2013. URL `http://winmerge.org/?lang=en`.

wxWidgets, 2016. URL `https://www.wxwidgets.org`.

Xmllint, 2015. URL `http://xmlsoft.org/xmllint.html`.

Christopher C. Yang. Fuzzy bayesian inference. *IEEE International Conference on Systems, Man, and Cybernetics*, 3:2707–2712, 1997. doi: 10.1109/ICSMC.1997.635347.