Clone Detection in Matlab Simulink Models

Hauke Petersen

Berlin 2012
IMM-M.Sc.-2012-02

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

IMM-Master: ISSN 0909-3192

Abstract

A growing amount of embedded software is created by automated code generation from models. As each development project requires a given level of software quality, it is essential for successful projects that a projects quality is monitored and assessed continuously during the development life cycle. While quality assessment tools and processes for conventional software engineering are widely available, quality assessment for models is a fairly young research topic. Clone detection in models is one method to assess and improve a model's maintainability. Matlab Simulink is one of the leading tools for model based software development in the automotive industry. In this thesis a novel clone detection algorithm is proposed. The algorithm was developed by analyzing the structure and characteristics of graph based models in general and Simulink models in particular. In contrary to existing clone detection algorithms the new algorithm follows a top-down approach for identifying potential cloned parts of a model. Further the proposed solution is prototypically implemented in a clone detection tool for Matlab Simulink models.

Contents

Chapter 1

Introduction

1.1 Motivation

Embedded systems can be found in a growing number of fields and applications. With an increasing use of micro-controller based platforms comes a higher demand for embedded software. Additionally a trend to substitute functionality formerly implemented in hardware with software-based systems developed over the last years. This is especially true for applications in the automotive industry. The 2007 BMW 7-series for example deploys a total of 67 embedded platforms running approximately 65 megabytes of binary code [23]. This software, particularly in safety critical contexts, needs to meet high quality standards when it comes to stability and reliability. For economic reasons there are also substantial requirements for maintainability as well as code size.

Beginning from control-, safety-critical and other embedded software it is becoming increasingly common to develop the software using model-based practices. In order to handle these complex software systems conventional software development processes are substituted by tools that enable developers to design systems on a higher abstraction level with the help of modeling languages and tools. These models are then used to automatically generate source or binary code. The model based approach has the further advantage that systems can be simulated and evaluated before even a single line of code has been written.

To develop software at a required quality level using minimal resources it is important to assess the software's quality as early as possible in the development process. Figure 1.1 shows a simplified example of such a process. In conventional software engineering, tool aided quality assessment can only be done in the implementation phase, for example by doing static code analysis. As the quality of code generated from a model is directly depending on the quality of that model, using a model-based approach enables developers to assess the softwares quality already in the design phase.

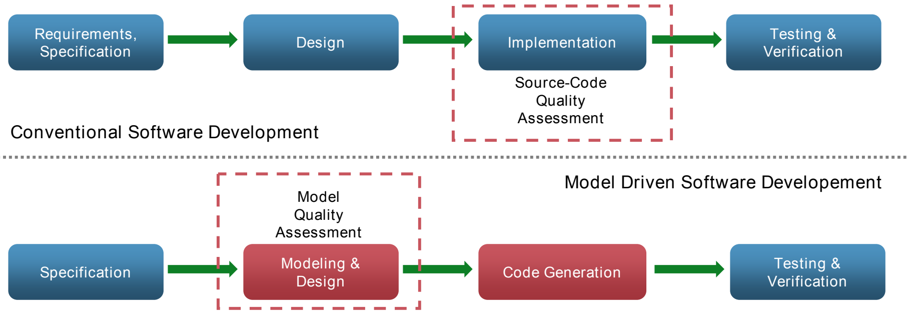


Figure 1.1: A simplified software development model. By using model-based development techniques, it is possible to evaluate a products quality earlier in the development process.

In the automotive industry it is common that software is used for a variety of car models and generations. Up to 90% of code can be reused from one vehicle generation to the next [23]. For this the software needs to be written in a way that it can be easily customized and reused. As code clones affect the re-usability in a negative matter it is important to identify them.

The results of a clone detection tool play a roll in two aspects of a development process: For one identified clones are a hint to developers that cloned parts can be combined into library functions. Secondly the resulting key numbers can be used in metrics as part of a quality model for assessing aspects of the model's quality as maintainability or re-usability.

1.2 Problem Description

Software clones can be defined as "segments of code that are similar according to some definition of similarity [17]". To transfer this to the scope of models a clone in a model could be described as at least to parts of the model, which are equivalent to some measure. A typical source of clones are copy and paste operations with subsequent more or less slight alternations of the copied part.

A large research problem is the definition of similarity measures for models. When looking for example at MATLAB Simulink models, they contain a lot of information which is not of interest to a developer. So when comparing two parts of a model, it is usually not important if they have different colors or if they contain different user comments. It might how ever be of importance if the two parts have the same semantical meaning and if they have the same internal structure. As stated by Deissenboeck et al. in [2] the definition of a suitable similarity measure is important to obtain relevant results.

Two parts of a model can look very different but they can still have the same functionality and meaning. This makes it hard to compare them and to find out if they are clones of each other. For Simulink models this problem can be generalized to the problem of graph isomorphism as Simulink models can be represented as graphs. In regard to complexity this problem is known to be in NP, but it is not known to be in P or to be NP-complete [7]. As Simulink models can become very large typical models are composed of 15.000 and more elementary building blocks [25]. An efficient solution to the subgraph isomorphism problem is essential to a scalable clone detection algorithm.

1.3 Objectives

The objective of this thesis is to find a clone detection algorithm that gives better results as existing solutions. The focus hereby is put on the completeness and relevance of the results. The algorithmic performance is not a primary concern for this work, it should however always be kept in mind.

A second goal is to design and implement a framework that can be used to compare and evaluate different solutions and approaches. The implementation should be loosely coupled so that an algorithm can be composed of different key elements and their impact can be evaluated. Further the implementation should have an easy to user interface so it can be included as a library into other software tools.

1.4 Overview

Chapter 2 gives a short overview on related work in this field. Known solutions and algorithms are presented. Further some terminology used in this document is defined.

Chapter 3 gives a brief introduction to MATLAB Simulink models and their structure. It is followed by a short analysis of the Simulink specific properties as well as a description on how the models can be represented in form of graphs.

In section 4 a novel clone detection algorithm is presented. The algorithm's concept is described in detail and a formal notation is given.

A prototypical implementation of the new algorithm is described in chapter 5. The basic data-structures and implementation decisions are pointed out in detail.

Chapter 6 describes aspects about visualization of results. Connected to this an interface to control Matlab from external Java programs is presented.

In chapter 7 the presented algorithm is compared to an existing solution. The results are further evaluated to identify shortcomings and strengths of the new approach.

Final conclusions and an overview on possible future work is given in chapter 8.

Chapter 2

Related Work and Terminology

Clone detection can be seen as a sub-field of quality assessment. In the following sections an overview on the current state-of-the-art in the fields of model quality and clone detection is given.

2.1 Software Quality

Quality can mostly not be put to a hard number, it can only be estimated. There are often domain or project specific reasons for using techniques that compromise conventional quality measures. The ISO 25010 standard defines a quality framework for measuring a software's quality. The standard defines a set of desirable characteristics which are functionality, reliability, usability, efficiency, maintainability and portability. Further the standard defines a list of measurable software attributes which contribute to one or more of the quality categories.

Measuring software attributes as for example the code complexity, the portability or the code documentation is done by applying metrics. These can be simple metrics as lines of code or the ratio of comment lines to code lines. They can also be of more complex nature as for example the McCabe or Halstead metrics. These and further common metrics are described in detail in [18]. The decision of which metrics are relevant can be varying for different projects. Keeping track of the chosen metrics during the development process can considerably increase a product's quality and help to decrease development costs.

An other key element for reaching a desired level of software quality is the use of programming and design guidelines. Additional to design guidelines that are defined for specific projects there exist generic guidelines for complete industry branches. Most software developed in the automotive sector is for example required to follow large parts of the MISRA guidelines [32].

The calculation and assessment of software metrics is part of most software engineering tools. In addition to these tools a group of programs exists, that can analyze code in more depth by doing static or dynamic code analysis. This way typical defects as range overruns, division by zero and similar can be detected. Leading tools for software quality assessments are for example QAC [21], Polyspace [28] and Sotograph [10].

2.2 Model Quality

With the spread of model driven engineering in electronic and software development the research field of model quality is gaining an increasing amount of attention in recent years. As in model based development the source code is automatically generate from models, it has been shown that the generated software's quality is directly proportional to the model's quality.

As for software, one problem that often leads to bad model quality is the fact, that modeling tools give a lot of design options to developers. The same task can be solved in many different ways. To force developers to solve problems in similar ways, there exist also design guidelines for Simulink models. The most important once for the automotive industry are MISRA for Simulink models [32] and the MAAB guidelines [31].

The leading approach to measure model quality is to transfer techniques and quality frameworks from conventional software engineering to model based engineering. Travkin and Stürmer present in [33] a tool for automated model quality assessment of Simulink models. Scheible et al. describe in [25] a generic quality model for measuring model quality following the characteristics given in the ISO 25010. As models have different characteristics than source code Holoch presents and evaluates a number of metrics that are suited to be used to evaluate a model's quality [11].

As maintainability plays an important role in the automotive industry Deissenboeck et al. present in [4] a quality framework that that focuses on improving a model's maintainability. Their approach is to keep track of defined attributes in parallel to the development so that parts of a model which are possibly bad to maintain can be redesigned early on.

2.3 Clone Detection

Clone detection for software is been a research topic for some time. Roy and Cordy give in [24] a detailed overview on different approaches and available tools. They also give the following classification of software clones:

Type 1: "Code is copied exactly except for variation in whitespace, layout and comments."

Type 2: "Code is syntactically identical except for variations in identifiers, literals, types, and variations permitted under Type 1."

Type 3: "Code is copied but can be further modified by changed, added, or removed statements, in addition to variations allowed under Type 2."

Type 4: "Code fragments undertake the same computation but using different syntax."

Gold et al. propose in [9] a classification framework to generalize the idea from software clones to visual dataflow languages in general. They define the following four groups of clones:

DF0: "Exactly-copied code fragments."

DF1: "Exactly-copied code fragments except for non semantics-affecting variations in layout and variations in comments."

DF2: "Exactly-copied code fragments except for non semantics-affecting variations in layout, variations in comments, and changes to literal values."

DF3: "Code fragments with modifications allowing additions, deletions, changes to connections, and free movement of objects."

One of the first algorithms for clone detection in Simulink models was presented by Deissenboeck et al. in [3]. The algorithm covers clones in the groups DF0 to DF2 and detects exact clones. The algorithm is used in the CloneDetective tool written by Juergens et al. [14]. CloneDetective is designed as an open source "workbench for clone detection research" which emerged into the open source tool ConQAT [26]. For this work it was decided to go with a new implementation from scratch instead of using the ConQAT tool. This was due to a lack of documentation and due to the fact that the results of this work should be independent so they can be used in tools of industry partners without any licensing problems.

In 2009 Pham et al. proposed an other clone detection framework for Simulink models called ModelCD [22]. The framework consists of two algorithms, eScan and aScan. While the eScan algorithm is designed to find exact clones of types DF0 to DF2, the aScan algorithm can also find approximate clones of type DF3. To improve the results gained by these algorithms Al-Batran et al. propose a notion to normalize model graphs using the models semantic information [1].

Hummel et al. present in [12] an approach to clone detection by storing indices of fragments of a model in a database. The approached is based on the idea that most of the times only small parts of a model are altered between clone detection runs. In their approach only newly changed parts of the model are taken into consideration in consecutive algorithm runs.

2.4 Graph Notation and Terminology

Simulink models can be represented as labeled sparse graphs. Such a model graph can be used as input for clone detection algorithms. For a consistent notation the following terminology similar to the one in [22] is used in this work. A model's representation is a labeled, directed sparse graph $G(V, E, L)$.

Definition 2.1 (Fragment) A fragment is any subgraph of G that consists of a set of connected vertices of G.

Definition 2.2 (Clone) Two fragments are considered to be a clone, if they are not overlapping and if they are similar in regard to a given definition of similarity.

Definition 2.3 (Clonegroup) A clonegroup is a group of fragments in which each two fragments are cloned.

When only looking at single vertices or edges of G, then two vertices or two edges are considered to be cloned if they have the same label.

2.5 The Graph Isomorphism Problem

When comparing two graphs with each other it can be hard to decide if they are identical. Figure 2.1 illustrates a small example of two graphs that have the same structure but look very different.

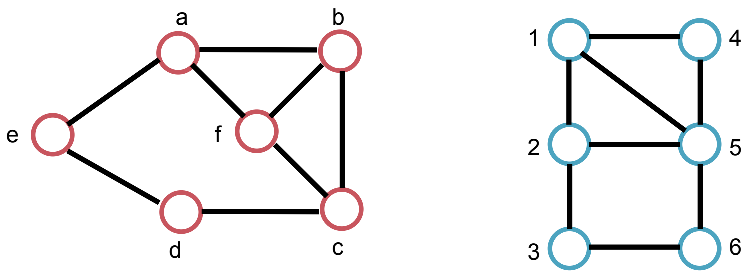


Figure 2.1: An example of the graph isomorphism problem: two graphs that look differently but do have the same structure.

This problem of comparing two graphs for similarity is known as graph isomorphism problem. The literature is not entirely sure in which complexity class this problem belongs. Fortin describes the isomorphism problem to be in NP, but it is not known to be in P or to be NP-complete [7]. Garey on the other hand states that the problem is NP-complete [8]. Over the years a lot of research was done on this topic. A very early algorithm to solve the subgraph isomorphism problem was presented by Ullmann in 1976 [34]. Messmer and Bunke propose in [19] an algorithm that can solve the isomorphism problem in polynomial time.

For clone detection the comparison of two subgraphs plays an important role as the comparison of subgraphs is one the key operations. In recent clone detection algorithms two solutions are being used: canonical labels and characteristic vectors. Both approaches produce a graph label that is equal for all possible isomorphisms of the graph.

A canonical graph label is produced by creating the graph's adjacency matrix and concatenating all entries of the upper triangle of this matrix to a string. This step is repeated for all possible permutations of the matrix, while the resulting strings are compared using their lexicographical order. As conical label the lexicographically smallest label is chosen. This label is equal for all isomorphisms of the graph. The problem with this approach is that the worst case computational complexity of this algorithm is $O(n!)$. But by using efficient heuristics and pruning techniques Junttila and Kaski show that the complexity can be significantly reduced [15, 16].

As canonical labels only work for exact clones Nguyen et al. present a labeling technique that can measure the similarity of graphs independent to their isomorphisms [20]. Their solution is called characteristic vectors. Such a vector is computed by saving two kind of key-value pairs in a vector: vertex labels and the number of their occurrence and the concatenation of vertex labels on all possible paths in the graph and the number of their occurrence. Taken for example a graph with two connected vertices that are labeled "a" and "b", the characteristic vector would be $[(a, 1); (b, 1); (ab, 1)]$. By ordering the resulting vector in lexicographical order by their key values two vectors can be compared by computing their vector distance.

Chapter 3

Analysis of Matlab Simulink Models

In this chapter the basic structure and properties of Matlab Simulink models are described. It is further analyzed how clones in such models can be defined and what they look like.

3.1 Introduction to Matlab Simulink

"Matlab is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numerical computation [27]". Matlab can also be used as computational back-end for a variety of specialized tasks. It can be extended by modules, so called toolboxes. There exist toolboxes for various applications as for machine learning, for different kind of statistics or for creating user interfaces.

Simulink is such a toolbox. It provides a graphical environment for designing, modeling and simulating dynamic systems [30]. Simulink can be used to model a wide variety of complex systems as for example motor control, access systems or computer vision algorithms. Simulink itself can further be expanded with other toolboxes to add domain specific functionality, as for example special tools for modeling real-time systems [29]. A very important aspect about Simulink models is that they can be used to automatically generate code for various embedded platforms. A leading tool for this is dSpace's Targetlink [5].

In Simulink models are created by using a graphical user interface. It can model continuous as well as discrete systems. Complex models are build of a small number of elementary building blocks which are connected with each other. Before a system can be simulated the model of the system has to be compiled. During this process Simulink compiles the model into an equation system that is then solved by Matlab. Simulink provides a number of solvers and parameters for this process so models can be simulated as required by a project.

Simulink models consist of elementary building entities, so called blocks. Blocks have different functionalities as for example basic mathematical functions (add, gain, multiply), boolean functions (and, or) or structural roles (switch, mux). There is also a set of user defined blocks whose functionality can be freely configured by the user. These are for example s- and m-function blocks which contain user written Matlab or C code. To be able to connect blocks with each each other they contain input and output ports. A block's number of ports can be fixed or variable depending on the blocks function. A gain block for example has always one input and one output port. A sum block on the other hand can have two or more input ports while always having one output port.

Figure 3.1 shows a simple Simulink model containing mostly math blocks. The model shown has no explicit purpose, it is rather used to clarify a typical Simulink model's structure. The model contains a number of source blocks, that are blocks that do not have any input ports. These are connected to some basic math blocks before the signal is send to a scope block which acts as a sink in the data flow.

Blocks are interconnected by signal lines. A signal line is a directed connection that transfers data from an output port to one or more input ports. Lines are allowed to have fan-outs so that there is a $1 : n$ relation between output and input ports. The data transfered on signal lines is variable. Signals can carry a single scalar value as well as n-dimensional vectors.

All blocks, ports and signals in a model have parameters. These parameters can be split into two groups: explicit and implicit parameters. Explicit parameters consist of values specified by the developer while designing the model. For blocks these parameters include for example the block's type, name and path, the number of ports or block specific parameters as the gain factor in a gain block. Block names have to be unique inside a subsystem while a block's path is unique in the whole model. The implicit parameters are generated when a

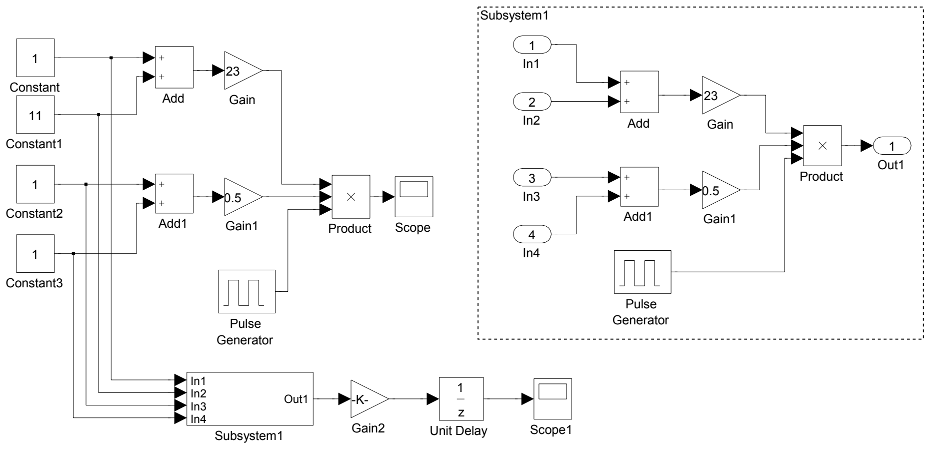


Figure 3.1: Sample Simulink model containing elementary math blocks and a small subsystem.

model is compiled. Implicit parameters can be for example the width of a signal line.

An important feature of Simulink is the possibility to structure models using subsystem and library blocks. This makes it possible to create very large and complex model without loosing the overview. This structural measures could be compared to the object oriented paradigm in conventional programming. By using subsystems models are partitioned into a layered hierarchy. Figure 3.2 shows an example of the subsystem hierarchy of a small model with four layers. The subsystem hierarchy can always be seen as a tree structure with the model itself as the tree's root. The tree's leafs can only be subsystems that do not contain any other subsystems.

A model's structure may be how ever not always as clear as the subsystem tree may infer. By the use of for example goto and from blocks signals can be connected across subsystem boundaries anywhere in the model.

3.2 Graph Representation of Simulink Models

Simulink models can be represented as directed, sparse graphs. In literature as in [22] or [3] slightly different graph representations and different labeling functions are used. As the structure of the used graphs is mostly similar the approaches for

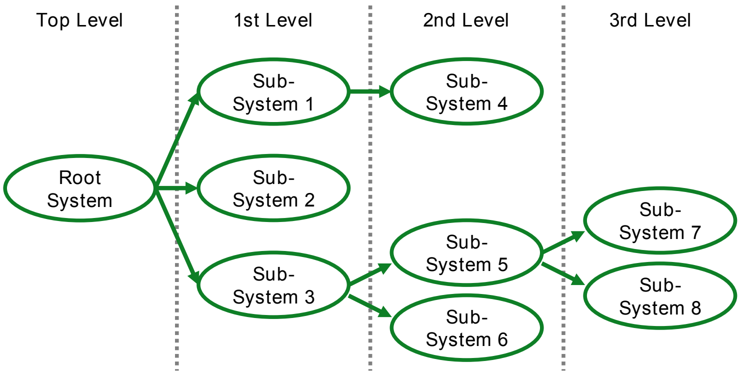


Figure 3.2: A tree visualizing the hierarchy of subsystems in a Simulink model.

labeling the graphs can substantially be different. In the following a description of a general graph representation as used in the following document is given.

For each block and each port of a model a vertex is created. Vertices representing ports and vertices representing blocks are then connected by directed edges in the direction of the data flow. Edges between input port vertices and block vertices are pointed towards the block vertex, edges between block vertices and output ports are pointed towards the output ports. Output and input ports that are connected via a signal line in the model are also connected by edges, pointing from output port vertices to input port vertices. As signal lines follow a $1 : n$ relationship, a precise graph representation would be a multi-graph. But for simplicity reasons the graph representation chosen is reduced to a standard graph by representing each signal line that has a fan-out with one edge for each input port that the signal is connected to. Figure 3.3 shows an example for a Simulink model and the corresponding graph representation.

As ports contain only a minimal amount of information the graph representation is further simplified by removing the port vertices from the graph and connecting block vertices directly with edges. As the removal of the port vertices leads to the loss of information, some information about ports can be kept by integrating it into block or signal labels. Figure 3.3 illustrates how the simplified graph of a small model looks like.

In the following document this simplified graph representation is denoted as model graph. Vertices of the model graph, as they represent blocks, are also called blocks while edges in the model graph are denoted as lines. This terminology is introduced for easier distinction between elements of the model graph and elements of a later introduced abstraction of the model graph.

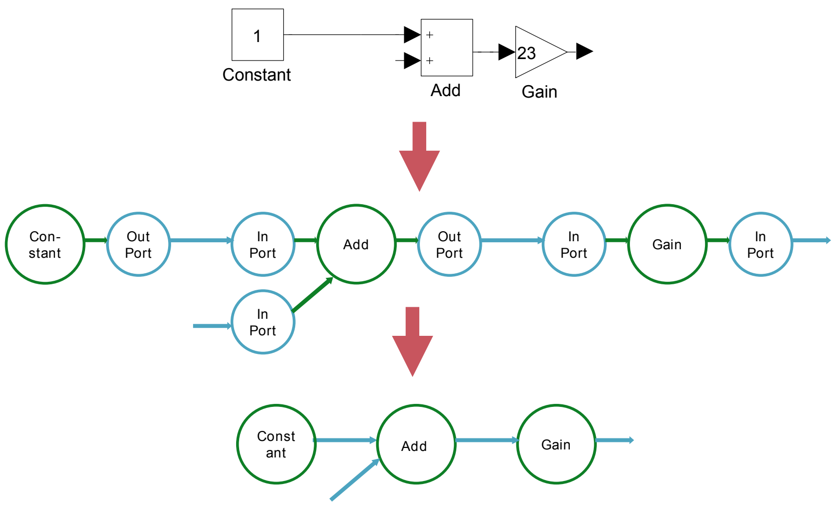


Figure 3.3: A generic and a simplified graph representation of a Simulink model. The lost port information in the general graph can integrated into vertex or edge labels of the simplified graph.

Labeling the model graph has a big influence on the results of clone detection algorithm runs. As vertices and nodes are compared by comparing their labels, the choice of a labeling function can significantly alter the results. To label the model graph all information of the underlying blocks and signal lines is available. There are how ever attributes that are more suited than others. If for example vertices are labeled with the path of the block they are representing, no clones would be found as the block paths are unique values and subsequently no two labels are the same. More details on graph labeling are given in the following section when looking at the properties of clones in Simulink models.

A special handling is proposed for subsystems. As shown in figure 3.2 Simulink models have a structural hierarchy which can be highly developed. It is not uncommon for large models to have a few hundred subsystems on 10 and more hierarchy levels [25]. By looking at this model structure every subsystem could be represented by it's own graph. But to be able to process the complete model with all the subsystems at once, the hierarchic structure is flattened into one graph. For this each subsystem graph is added to the highest layer graph. The result of this is a graph that contains a number of connected subgraphs. To still be able to use the models structure, additional to the model graph a subsystem tree is created, in which the nodes are linked to subgraphs in the model graph.

3.3 Clones in Simulink Models

When looking at Simulink models clones can be defined in various ways. In section 2.3 clones are defined to be parts of a model that are similar to some similarity measure. This becomes very clear for the case that only elementary blocks are considered. It seems for example to be obvious that a constant block is not a clone of a sum block. The same way one would say intuitively that an add block is not the clone of a gain block. But on the other hand one could define the latter example very well as a clone for the case that clones are defined to be blocks from the same group as for example math blocks, logic blocks, source blocks.

For defining clones in Simulink models it is important to keep in mind what the clone detection results are used for. Looking at a model's maintainability the goal is to identify duplicate parts in the model that can be consolidated into library parts to increase the re-usability and reduce the model's size. Such a substitution is useful only for the case that the duplicate parts of the model have the same semantic meaning. Looking at elementary blocks this implies that it makes only sense to consider two blocks as cloned if they have the same functionality.

In the following document the block type is defined to be the similarity measure of elementary blocks. This results in a straight forward labeling function for the model graph: each vertex is labeled with the block type of the block it is representing. Figure 3.4 illustrates two cloned fragments in two different subsystems. As the example is kept very simple, it gives a good example what a clone would look like.

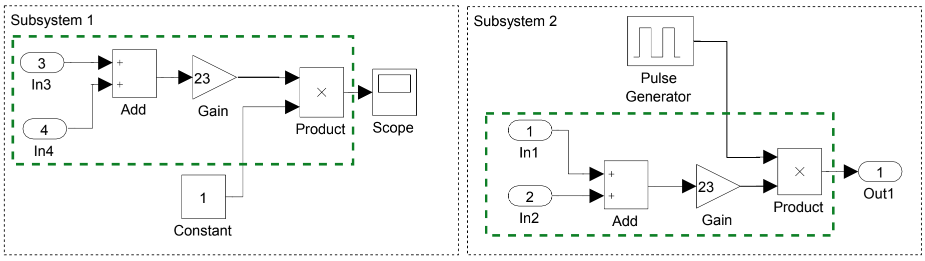


Figure 3.4: Example of an exact clone in a Simulink model. The blocks in the green dashed boxes form cloned fragments.

For developers that are analyzing the clone detection results not all clones are relevant. In this document two assumptions are made:

1. Subsystems are considered reasonably small, so that a developer can oversee the complete subsystem. So two cloned fragments inside a single subsystem are not relevant.

2. A clone detection algorithm should find cloned fragments as large as possible, because larger fragments have a higher chance to contain logical functionality that makes sense to put into a library block.

CHAPTER 4

A New Clone Detection Algorithm

In this chapter a novel clone detection algorithm for finding exact clones is proposed. In contrast to previous algorithms the new algorithm follows a top-down approach to generate clone candidates.

4.1 Design Considerations

A clone detection algorithm has to fulfill two basic properties to be useful: it has to find relevant clones as complete as possible and it needs to be scalable. As mentioned earlier, the relevance of cloned fragments can be hard to argue about. The relevance might even be perceived differently for developers depending on their intentions. As the scalability property implies that the algorithm needs to be able to process large models in a feasible amount of time, it can also depend on the usage of the algorithm how much time is considered as feasible.

One important application for clone detection is to find duplicate parts in a model which could be joined into library blocks. This not only improves the re-usability but also improves the maintainability of a model. For identifying these parts in a model, it is of advantage to identify fragments that are as large as possible. This way a fragments size will increase the relevance of a clone. Further it is beneficial to identify cloned parts that are not obvious to the developer, as for example entire subsystems that are used multiple in the model.

The scalability of the algorithm is important as models are becoming more complex and thereby their size is increasing. The algorithm should be able to handle very large model in a short time. For a developer it makes a significant difference if one has to wait a few minutes or a full day for results. For the case that the clone detection algorithm should be part of quality assessment framework, short running times would be a big advantage. An assessment tool can be run to analyze a model frequently during the development process with giving direct feedback.

The algorithms used in the ModelCD and CloneDetective tools [22, 3] basically follow both the idea of extracting small cloned fragments and expanding them edge by edge. Even with good heuristics this approach leads to a high number of candidate fragments that are being created. This also leads to a high number of small cloned fragments in the results, that may not be relevant. As comparing two fragments with each other is the most costly operation in these algorithms, a large number of generated candidate fragments further leads to high computational costs. The proposed algorithm follows a top-down approach to candidate fragment generation. It starts by creating only a limited number of larger fragments and then matches them against each other to find if eventually cloned core fragments of them.

4.2 Preprocessing

To apply the clone detection algorithm and to ensure that the algorithm yields optimal results, a few preprocessing steps are necessary. As first step the input model graph is transferred into another form to increase the differentiation between it's vertices. As second step the new graph is labeled using supplied labeling rules. These preprocessing steps require two thins as input data, the model graph and the hierarchy tree. The latter one is needed for a special handling of the models subsystems.

A key element in the clone detection process is to compare different fragments with each other to find out if they are equal or similar. This problem can be broken down to the concept of graph isomorphism. Possible solutions are for example the use of canonical labels or characteristic vectors. Both of them have in common, that the computational complexity grows with the number of vertices in a fragment, that have the same label. To increase the differentiation between vertices, an alternative graph representation is proposed, in the following called abstract graph.

Creating an Abstract Graph

For the new graph representation, the role of vertices and edges are switched. For each edge in the model graph, a vertex is created in the new graph. Further further for each edge in the model graph that is connected to the same vertex, an edge between corresponding vertices in the new graph is created. All newly created vertices are for later referencing linked to the underlying edges in the model graph. During this process, a copy of the subsystem hierarchy tree is updated to point to subgraphs in the newly created graph. Figure 4.1 shows an example for a conversion from the model graph to the new abstract graph.

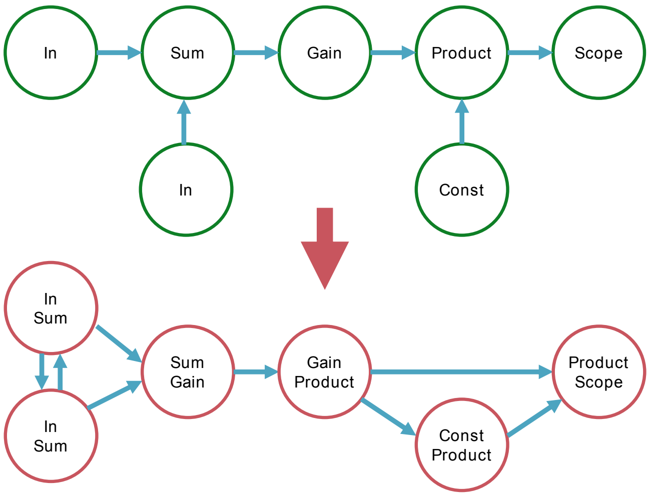


Figure 4.1: Transformation from the model graph representation to the abstract labeled graph.

Numbers of test graphs show, that the total number of vertices and edges in the abstract graph is slightly higher than the number of vertices and edges in the model graph. But as shown in the following, the disadvantage of working with a larger graph is highly compensated.

Labeling the Abstract Graph

Elementary blocks are labeled using their block type. As vertices of the abstract graph are linked to edges in the model graph, such a vertex can be further associated with exactly two blocks in the model graph, the underlying edges source and destination blocks. The label for a vertex is now created, by concatenating the labels of these two blocks, taking the label of the source block and appending the label of the destination block.

As taking the block type as a label works well for most elementary blocks, it does not suffice for blocks containing subsystems. Subsystem blocks should only be assigned equal labels for the case that their underlying subsystems are equal. For this reason subsystem blocks are not labeled with their block type, their label is computed by looking at the subsystem that the block contains.

When looking at the subsystem hierarchy, it is obvious that subsystem labels can only be created for subsystems that contain no other subsystems or for those that contain subsystems which are already labeled. For this reason the order in which the subsystem labels are created is given by a depth-first traversal of the subsystem hierarchy tree.

By creating the abstract graph direction information from the model graph is lost, because edges in the abstract graph are not distinct. This is how so ever not tragic, as for one some direction information can be still gained from the labels directly - the order in which the basic labels are concatenated. And for two the direction information is not superficially important to the clone detection algorithm, here the neighborhood relationship of two vertices is the crucial property.

For the new algorithm a simple graph labeling function is proposed. The function is a simplification of the characteristic vector given in [20]. It contains the following data concatenated to one string: The number of vertices in the graph and a lexicographically ordered list of tuples consisting of vertex label and the number of occurrences of this label in the graph. The list of label tuples is hereby ordered alphabetically by the labels. So an example label could look like the following: "7;Add:4;Gain:2;Product:1".

4.3 Algorithm Description

The new algorithms general structure of the algorithm can be split into two parts, preprocessing and labeling the input model graph and doing the actual clone detection on the labeled graph. The algorithm works with four input parameters:

1. A labeled graph representing the model $G(V, E, L)$.
2. The models subsystems as labeled fragments S.
3. A labeling function $L()$, that computes labels for fragments.
4. The minimum size of a cloned fragment s_{min}.

The input graph is supposed to have a structure as described in 3.2. The labeling function could be using for example canonical labels or characteristic vectors, but for this algorithm a simplified version of characteristic vectors is used that yields similar results.

The output of the algorithm is a set of clonegroups. Each clonegroup is a collection of fragments that are all clones of each other. Algorithm listing 1 shows the formal notation of the algorithm.

Algorithm 1 Novel Clone Detection Algorithm

```
procedure NCD($G(V, E, L), S, L(), s_{min}$)
    $Tmp \leftarrow \emptyset$
    $Touched \leftarrow \emptyset$
    $F \leftarrow extractCandidates(G) \cup S$
    for all $f_1 \in F$ do
        $Touched \leftarrow f_1$
        for all $f_2 \in F$ do
            if $f_2 \cap Touched = \emptyset$ & $L(f_1) \neq L(f_2)$ then
                $Tmp \leftarrow Tmp \cup findClonedCores(f_1, f_2, L, s_{min})$
            end if
        end for
    end for
    $CG \leftarrow filterAndGroup(F \cup Tmp)$
    return $CG$
end procedure
```

The core algorithm concept works on all generic labeled, directed graphs. However the described design is tailored to work on Simulink models using some domain specific properties for labeling and subsystem handling.

4.3.1 Candidate Generation

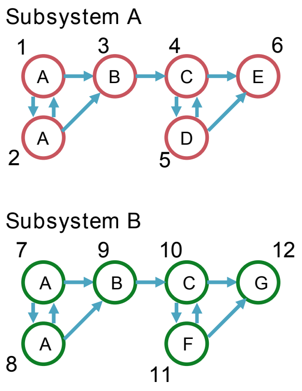
The key element of the proposed clone detection algorithm is the extraction of clone candidate fragments. These are fragments of the graph that have a high chance of being at least partially cloned. The candidate fragments are identified by analyzing a special form of the input graph's adjacency matrix.

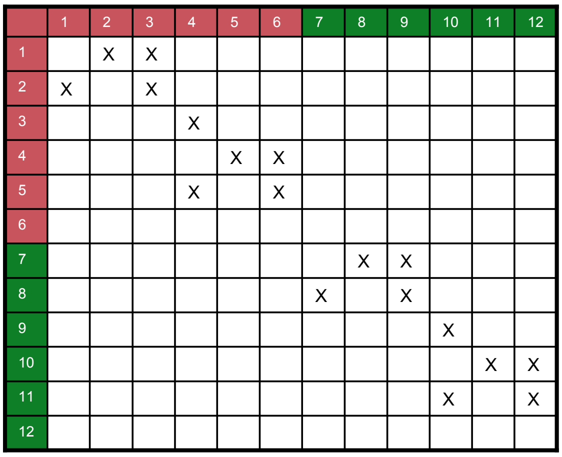
Extracting candidate fragments from the input graph is based on an elementary idea. Let there be a small fragment that consists only of two vertices connected by an edge. Then this fragment can only be part of a clone if there is another fragment of the same size that contains two nodes with identical labels than the first one. The candidate generation process is now extracting all small fragments of that size and joins them together to form as large as possible candidate fragments.

A commonly used graph notation is to represent a graph in form of an adjacency matrix. A graph with vertices $v_0, \ldots, v_n$ is noted as square $m \times m$ matrix with $m = (n + 1)$. In the simplest case each field $a_{i,j}$ of the matrix is set to 1 if there is an edge connecting v_i and v_j or a 0 if no edge exists between the two vertices. Figure 4.2 gives an example of a small input graph and the corresponding adjacency matrix. Instead of just using ones and zeros it is also possible to store more information about the edges in the matrix. For example the edge's labels could be used.

For finding small cloned fragments a new adjacency matrix representation is proposed. For this the adjacency matrix described is compressed by forming a $c \times c$ matrix where c is the number of distinct vertex labels in the input graph. This means that instead of having a column and a row for each vertex of the input graph the compressed matrix has only one column and row for each vertex label that is used in the input graph. Given the case that no label occurs twice in the input graph, the compressed matrix has the same size than the adjacency matrix.

When compressing the matrix a second alteration is made to the notation. As the compressed matrix is now showing the connections between groups of vertices with the same label, the connections are not marked by a one or a zero. Instead for each connection between two groups a tuple (g_1, g_2) of two integers is saved. Such a tuple has the meaning that from g_1 different nodes in one group there are edges to g_2 different nodes in another group. Let there be a tuple $(2, 2)$ in the matrix at the position $a_{i,j}$. Then the two numbers in that tuple say, that





	1	2	3	4	5	6	7	8	9	10	11	12
1		X	X									
2	X		X									
3				X								
4					X	X						
5				X		X						
6												
7								X	X			
8							X		X			
9										X		
10											X	X
11										X		X
12												

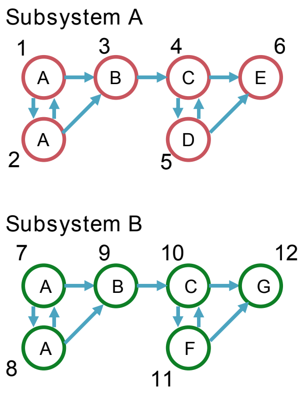
Figure 4.2: The adjacency matrix for a model with two subsystems. The numbers are unique identifiers for the vertices, the letters are the vertices labels.

there are connections from two different vertices in the group i to two different vertices in the group j. The compressed adjacency matrix for the graph shown in figure 4.2 is illustrated in figure 4.3.

The tuples in the compressed matrix carry a simple statement. Only if both numbers in the tuple are ≥ 2 the vertices represented by the tuple are possibly part of a cloned fragment. If at least one of the two numbers is ≤ 1 then the vertices represented by that tuple are not of any interest. For this reason all vertices that are represented by tuples with both numbers ≥ 2 are saved in a list. To finally extract large candidate fragments, fragments are build by expanding the vertices in this list recursively with neighboring vertices that are also part of the list. All newly created fragments that are larger than the given minimal fragment size s_{min} are finally combined with the set of subsystem fragments as these are also considered as candidate fragments.

4.3.2 Finding Cloned Cores

The described candidate generation algorithm extracts potential cloned fragments. These fragments are how so ever mostly not directly clones of each other. But as the basic idea behind the fragment extraction was to only consider parts of the input graph that are definitely occurring in the graph more than once the extracted fragments have a tendency to have cloned cores.



	A	B	C	D	E	F	G
A	4-4	4-2					
B			2-2				
C				1-1	1-1	1-1	1-1
D			1-1		1-1		
E							
F			1-1		1-1		
G							

Figure 4.3: The compressed adjacency matrix for the same model as in figure 4.2. The tuples in the non-empty fields of the matrix mean, that there is a connection from x distinguished vertices of the first group to y distinguished vertices in the second group.

When looking for cloned cores, each fragment is compared to each other fragment in the list of candidate fragments. Two fragments are however only to be considered for having cloned cores if they have different graph labels. If they have the same graph label they are already clones of each other. Listing 2 shows the algorithm used for finding cloned cores.

The identification of these cloned cores is done in a rather naive way, which could be an entering point for further improvement of the algorithm in the future. Let f_1 and f_2 be two fragments, for which is true that $sizeof(f_1) \leq sizeof(f2)$. The approach to finding cloned cores is done in the following way: first f_{c2} is created as a copy of f_2. Next all vertices from f_{c2} are removed, that have labels which are not present in f_1. If the size of f_{c2} is less than the minimum fragment size, no cloned core is present and further comparison is skipped. Otherwise the label for f_{c2} is calculated and compared to the label of f_1. If the labels are the same it means that a clone of f_1 was part of f_2. For that is the case, f_{c2} is returned. If that is not the case but $sizeof(f_{c2} \geq s_{min}$, f_{c1} is created as copy of f_1. From the fragment f_{c1} are then in the same way than before all vertices removed, that have labels which are not present in f_{c2}. If f_{c1} is then still $\geq s_{min}$, the label for f_{c1} is computed and compared to the label of f_{c2}. For the case the labels are equal, both new fragments f_{c1} and f_{c2} are returned.

Algorithm 2 Finding Cloned Cores of two Fragments

```
1:  procedure findClonedCores($f_1, f_2, L, s_{min}$)
2:      ASSERT: $sizeof(f_1) \leq sizeof(f_2)$
3:      $f_{c2} \leftarrow copy(f_2)$
4:      $f_{c2} \leftarrow strip(f_1)$
5:      if $f_{c2} \geq s_{min}$ then
6:          if $L(f_1) == L(f_{c2}$ then
7:              return $f_{c2}$
8:          else
9:              $f_{c1} \leftarrow copy(f_1)$
10:             $f_{c1} \leftarrow strip(f_{c2})$
11:             if $f_{c1} \geq s_{min}$ & $L(f_{c1}) == L(f_{c2})$ then
12:                 return $f_{c1}, f_{c2}$
13:             else
14:                 return $\emptyset$
15:             end if
16:         end if
17:     end if
18: end procedure
```

4.3.3 Grouping and Filtering

The final step of the clone detection algorithm is to filter and group the extracted fragments. This is done by iterating threw all found fragments and putting them into groups by their label. Due to the way the candidate fragments are created, it is possible that there are fragment which cover identical vertices. To remove these duplicates from the result clonegroups, each fragment in a clone group is checked if it overlaps with any other fragment in it's clonegroup. If that is the case, the fragment is removed from the group.

As a final clean-up step every clonegroup that contains less than two fragments is removed.

CHAPTER 5

Implementation

In this chapter it is described how the proposed algorithm was implemented into a small clone detection tool. The basic design decisions are explained as well as some implementation details are given.

5.1 Clone Detection Tool Architecture

The proposed clone detection algorithm is implemented in a clone detection tool to prove the algorithms concept as well as to create a sample implementation of the algorithm which can be re-used in other software tools. The main design goal for the implementation is to structure it in a framework like fashion, which makes it easy to be adapted to various environment and which makes it easy to customize parts of the tool. This thoughts lead to a list of key requirements:

- The implementation should be as modular as possible.
- The different parts of the implementation need to have clearly defined interfaces.
- The implementation should be able to be run stand-alone or to be used as a library within other programs.

The whole clone detection process can be seen as a data processing pipeline. This pipeline consists of four major steps: Parsing a model, preprocessing the model, running the actual clone detection algorithm and presenting the results. The input of the pipeline for the proposed tool is a Simulink model, the output is a list of clonegroups found in this model. Figure 5.1 illustrates this pipeline.

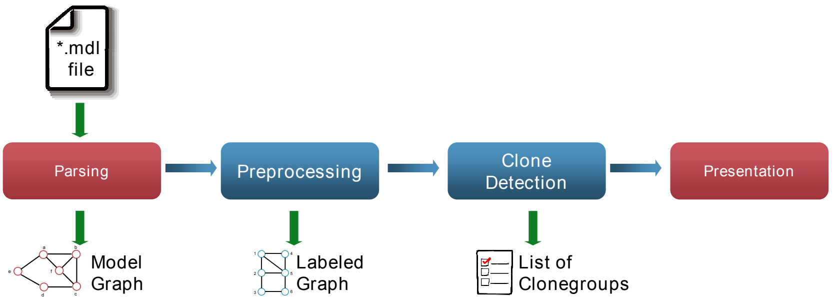


Figure 5.1: The clone detection pipeline implemented in the proposed tool.

The data-flow threw the described pipeline is rather straight forward. The parser stage reads the input Simulink model and builds a model graph as described in section 3.2. The next stage reads the model graph and builds a more abstract graph representation of it. The abstract graph used for this implementation is as described in section 4.2. It is also the pre-processing stage's job to label the abstract graph. This step is in so far important, as it defines which elementary nodes of the graph will be regarded as clones by the following clone detection stage. This stage applies the actual clone detection algorithm. The result of the clone detection run is a set of clonegroups which were discovered in the given abstract graph. The presentation stage is finally responsible for displaying the clone detection results. This is described detailed in chapter 6.

When looking at the interfaces between the different pipeline stages, it is obvious that a single stage can substituted, as long as the substitution uses the same kind of data structures. This makes it possible to create for example different kind abstract labeled graphs during the pre-processing stage and feed them to the same clone detection algorithm without making any changes to the latter. This enables the implemented tool to be adaptable to any kind of external requirements. It makes it further easy to use the tool with different algorithms or labeling functions to compare their performance regarding the clone detection results.

5.2 Platform and Tools

The main motivation to this work given by the industry partner was to integrate the clone detection algorithm into a model quality assessment tool. Since all prototypic implementations by the industry partner are using the Java based Eclipse Rich Client Platform [6], the clone detection tool is also developed using the Java programming language in version 6. Choosing Java makes it further possible to develop a stand-alone clone detection tool for proof-of-concept and benchmarking runs and integrate the core algorithm any time later to existing or new tools.

The Java programming language is further known for a very library support. The build-in standard library already supports the most commonly used data-structures with a lot of support functionality. Java developers can further choose out of a wide variety of existing and freely available libraries under various software licenses. The presented tool makes heavy use of Java's standard HashMaps and the related HashSets. These data-structures have the advantage, that the basic put and get operations are carried out in constant time in case the stored objects have a differentiating hash value.

For parsing Simulink models, the tool makes use of the Simulink model parser and builder used in the ConQAT tool [26]. Since the source code of the ConQAT tool is freely available under an open-source license, the model parser was extracted from it and used in the presented tool. The parser works by reading **.mdl* files, which is the standard format that Simulink uses for saving models. The advantage of this approach is, that it works independent of any local Matlab installation, so the parser can be used if there is no Matlab installation on the same machine. The parser is this way further able to parse models which can not be compiled by Simulink, which could have various reasons from design faults to wrong configurations or missing library files.

Despite the described positive attributes, the parser suffers of some shortcomings. The biggest problem is, that the parser can only handle Simulink blocks, that are hard coded into the parsers code. This works fine for most Simulink models that exist of blocks from the standard libraries. But it can lead to crashes if models contain blocks that are not known to the parser. A second shortcoming of the parser is, that by parsing the static *.mdl model files, the parser has only access to the model's explicit parameters. The parser can not access any parameters, that Simulink creates while compiling a model. Such parameters can be for example the dimension of a signal line.

5.3 Internal Graph Implementation

As the clone detection algorithm is heavily depending on the graph it is working on, the graph representation plays an important role for the implementation the clone detection pipeline described earlier. For a consistent graph access and handling, two global graph interfaces are defined: *IGraph* and *INode*. The graph interface is used for all classes representing graphs or subgraphs, the node interface is used to define the handling of vertices. The full interface definitions are given in the UML diagram in figure 5.2.

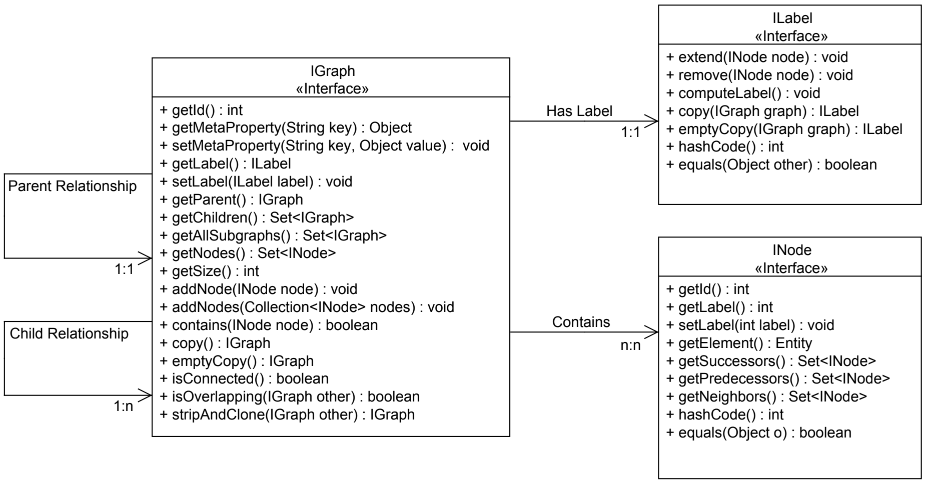

IGraph
«Interface»
+ getId() : int
+ getMetaProperty(String key) : Object
+ setMetaProperty(String key, Object value) : void
+ getLabel() : ILabel
+ setLabel(ILabel label) : void
+ getParent() : IGraph
+ getChildren() : Set<IGraph>
+ getAllSubgraphs() : Set<IGraph>
+ getNodes() : Set<INode>
+ getSize() : int
+ addNode(INode node) : void
+ addNodes(Collection<INode> nodes) : void
+ contains(INode node) : boolean
+ copy() : IGraph
+ emptyCopy() : IGraph
+ isConnected() : boolean
+ isOverlapping(IGraph other) : boolean
+ stripAndClone(IGraph other) : IGraph
Parent Relationship
1:1
Child Relationship
1:n
Has Label
1:1
Contains
n:n
ILabel
«Interface»
+ extend(INode node) : void
+ remove(INode node) : void
+ computeLabel() : void
+ copy(IGraph graph) : ILabel
+ emptyCopy(IGraph graph) : ILabel
+ hashCode() : int
+ equals(Object other) : boolean
INode
«Interface»
+ getId() : int
+ getLabel() : int
+ setLabel(int label) : void
+ getElement() : Entity
+ getSuccessors() : Set<INode>
+ getPredecessors() : Set<INode>
+ getNeighbors() : Set<INode>
+ hashCode() : int
+ equals(Object o) : boolean

Figure 5.2: UML diagram of the used graph interface definition.

Edges in the graph are not given explicitly, they are defined by the way the node objects are referencing each other. By calling the *getPredecessors()* or the *getSuccessors()* methods on a vertex, all vertices are returned, that are connected to that node with incoming or outgoing edges, respectively. The given graph notation offers also the shortcut method *getNeighbors()*, which offers a shortcut to querying all neighboring vertices. By using the *getNeighbors()* method, a graph can be traversed, if it would be an undirected graph. This property is used for efficiency reasons on some places in the implementation.

The graph interfaces *getChildren()* and *getParent()* methods are used to model a graph hierarchy. This is useful, when the relationship between a graph and it's subgraphs needs to be saved. For the clone detection algorithm this construct is used to model the hierarchy between the subsystems in the model graph. Further it proves useful to note the relationship between candidate fragments and their extracted cores.

A special role in the graph interface definition plays the *ILabel* interface. This interface defines the methods that need to be implemented by graph labels. By only loosely coupling the graph label to a graph, it is possible to quickly exchange the graph labeling function so that it could be parametrized.

5.4 Implementation Details

To reflect the design considerations made earlier in this chapter, the implementation makes heavy use of the factory design pattern. As stated is the implemented tool's functionality is split into pipeline stages to allow exchanging and customization to selected processing steps. The factory pattern allows to to dynamically choose the single stages functionality at run-time. This allows for example to make to runs of the clone detection algorithm with different labeling functions without having to restart or recompile the Java program.

The run-time configuration for the processing pipeline is done with with four different object factories:

ModelFactory: Runs the ConQAT parser on a specified *.mdl file and returns the model graph.

GraphFactory: Transforms a given model graph into a specified abstract graph.

LabelFactory: Assigns labels to the vertices of the abstract graph based on the underlying model graph and the specified labeling function.

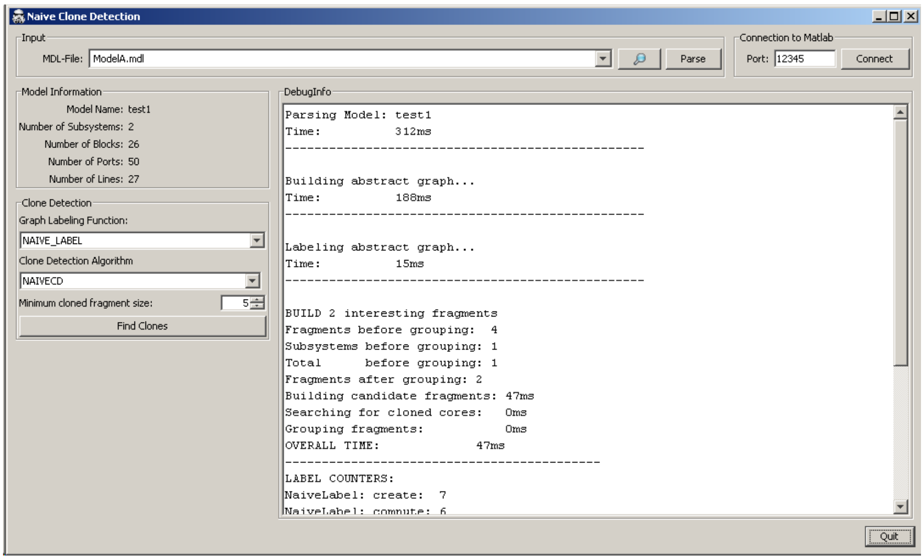
AlgorithmFactory: The algorithm factory returns a runnable Java object of the actual clone detection algorithm. The algorithm object further contains the clone detection results after a successful run.

All object factories accept parameters that can alter their behavior. When creating the label factory for example, it can be defined which kind of graph labels should be used, this can be canonical labels, naive labels or characteristic vectors. This concept proofs especially useful for further extensions of the tool. New functionality can be added to a corresponding factory, while all others parts of the program do not need to be changed.

As the computation and comparison of the graph labels takes the most processing time during a clone detection run, the implementation of the labeling function was optimized. The labeling functions are implemented using a lazy computation paradigm: the labels are only computed or recomputed when they are actually read. Depending on the labeling function, this can save a great deal of redundant computations.

For a further optimization of the labeling functions the fact is used, that graph labels are more often compared than they are computed. This means that the actual computed label can be stored in an indexed lookup table. By only comparing the indices instead of full labels, the comparison complexity is reduced from comparing two strings to comparing two integers. A graph label's index can further be used as a hash code of that label, which makes it possible to efficiently use graph labels as key value in HashSets and HashTables.

Figure 5.3 shows the main screen with available options of the implemented tool.



MDL-File: ModelA.mdl
Port: 12345

Model Name: test1
Number of Subsystems: 2
Number of Blocks: 26
Number of Ports: 50
Number of Lines: 27

Graph Labeling Function: NAIVE_LABEL
Clone Detection Algorithm: NAIVECD
Minimum cloned fragment size: 5

```
Parsing Model: test1
Time:           312ms
-------------------------------------------------


Building abstract graph...
Time:           188ms
-------------------------------------------------


Labeling abstract graph...
Time:           15ms
-------------------------------------------------


BUILD 2 interesting fragments
Fragments before grouping:  4
Subsystems before grouping: 1
Total      before grouping: 1
Fragments after grouping: 2
Building candidate fragments: 47ms
Searching for cloned cores:   0ms
Grouping fragments:           0ms
OVERALL TIME:             47ms
-------------------------------------------
LABEL COUNTERS:
NaiveLabel: create:  7
NaiveLabel: compute: 6
```

Figure 5.3: The main screen of the implemented clone detection tool. The GUI allows to chose an input Simulink file and start a clone detection algorithm with selected options.

Chapter 6

Results Presentation

For clone detection to be useful, the results need to be visualized in a productive way. As described in chapter 1, clone detection results are used for two scenarios: as part of metrics to measure model quality and for developers to improve the readability and maintainability of their models. In the following chapter the result presentation part of the implemented tool is described.

6.1 Result Structure

The result of the clone detection algorithm described and implemented in this work is a list of clonegroups. A single clonegroup consists of a set of fragments, of which each two fragments are cloned. A fragment consists further of a set of connected nodes in the clone detection algorithm's input graph. As the algorithm's input graph is an internal abstraction away from the model graph, a set of nodes from the input graph is not useful to a Simulink developer. It is therefore part of the presentation part, to link the results back to the graph model and further onto the Simulink model itself.

For a developer, two kind of information from the result are important: a list of statistical data that supplies information about the model's quality and the form and location of the cloned fragments inside the model. When looking at statistical data, the following key numbers are commonly taken into consideration:

# of CG: The number of clonegroups found.

Avg. group size: The average number of fragments in a clonegroup.

# of clones: The overall number of found cloned fragments.

Avg. clone size: The average size of all found cloned fragments.

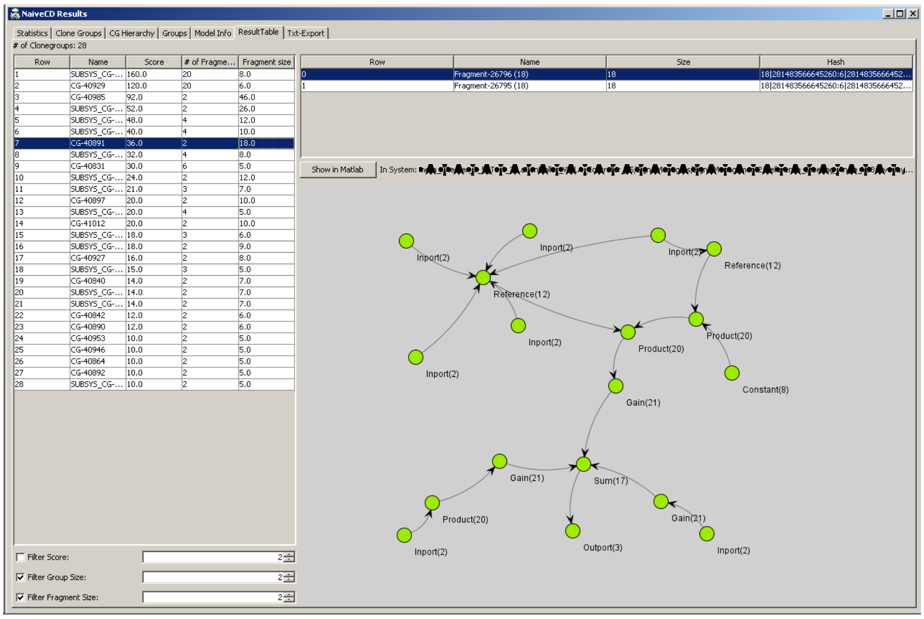
The proposed tool is computing the average numbers by taking the arithmetic mean. For having a more complete picture, the data can easily be extended by counting for example maximum and minimum numbers.

For the second scenario of using the results, a presentation of the results is required, which enables a Simulink developer to easily and fast browse the found clonegroups. As stated earlier, for some cases it is not possible to assess whether a clonegroup is relevant. For this reason developers need to find clonegroups relevant to them in an easy way. When looking at the fragments of a single clonegroup, the most important information is how may fragments there are in the clonegroup, part of which subsystems they are and which blocks they cover.

6.2 Result View

The visualization part of the proposed tool is split into two functionalities. The first step is to present the results in a way that the key numbers can be easily captured by a user. For this reason the result's key numbers are presented in form of two tables. The first table lists all found clonegroups and their characteristic numbers. The second table lists on selection of a clonegroup in the first table the fragments contained in the selected group. Figure 6.1 shows a screenshot of the result screen.

As the plain numbers for them self are not easy to assess for a user, the result view is further enhanced with a quick visual preview function. For this a selected fragment is plotted as a graph directly in the GUI. For this the Jung graph library [13] is used. Plotting the preview graph helps a user to estimate the character of a detected fragment. But as the graphing library layouts the preview graph in a rather random manner it can be at some times still hard to conclude from the preview graph to the original part in the actual Simulink model.



NaiveCD Results

# of Clonegroups: 28

Row	Name	Score	# of Fragme…	Fragment size
1	SUBSYS_CG-…	160.0	20	8.0
2	CG-40929	120.0	20	6.0
3	CG-40985	92.0	2	46.0
4	SUBSYS_CG-…	52.0	2	26.0
5	SUBSYS_CG-…	48.0	4	12.0
6	SUBSYS_CG-…	40.0	4	10.0
7	CG-40891	36.0	2	18.0
8	SUBSYS_CG-…	32.0	4	8.0
9	CG-40831	30.0	6	5.0
10	SUBSYS_CG-…	24.0	2	12.0
11	SUBSYS_CG-…	21.0	3	7.0
12	CG-40897	20.0	2	10.0
13	SUBSYS_CG-…	20.0	4	5.0
14	CG-41012	20.0	2	10.0
15	SUBSYS_CG-…	18.0	3	6.0
16	SUBSYS_CG-…	18.0	2	9.0
17	CG-40927	16.0	2	8.0
18	SUBSYS_CG-…	15.0	3	5.0
19	CG-40840	14.0	2	7.0
20	SUBSYS_CG-…	14.0	2	7.0
21	SUBSYS_CG-…	14.0	2	7.0
22	CG-40842	12.0	2	6.0
23	CG-40890	12.0	2	6.0
24	CG-40953	10.0	2	5.0
25	CG-40946	10.0	2	5.0
26	CG-40864	10.0	2	5.0
27	CG-40892	10.0	2	5.0
28	SUBSYS_CG-…	10.0	2	5.0

[ ] Filter Score: 2
[✓] Filter Group Size: 2
[✓] Filter Fragment Size: 2

Row	Name	Size	Hash
0	Fragment-26796 (18)	18	18|281483566645260:6|2814835666452…
1	Fragment-26795 (18)	18	18|281483566645260:6|2814835666452…

In System: [illegible]

Figure 6.1: Screenshot of the quick-preview functionality of the implemented tool.

To encounter this problem a second functionality for visualizing fragments in included in the tool. When a fragment is selected the user has the option to highlight the fragment directly in Simulink. For the implementation of this functionality a Java Matlab connector was implemented. The clone detection tool uses this connector to communicate directly with a running Simulink instance. This feature is how ever only available for users that have a running Simulink instance on their machine.

6.3 Matlab Connector

For developers it can be very useful to highlight fragments that were identified as clones inside Simulink in the original model. As Matlab provides no official interface to invoke commands from external programs, a generic Java tool was written.

Matlab is able to run Java code inside it's environment and is for that case running an internal Java Virtual Machine (JVM). Also there exists a non-documented Java interface for invoking Matlab commands from the internal JVM called Java Matlab Interface (JMI). The setback is, that Matlab commands have to be run from the main dispatcher thread of the internal JVM, so it is not possible to call JMI function calls from external Java programs running in different JVMs.

As solution for this problem, a Java Interface was written, that exists of two parts: A frontend library that can be included in any program that wants to invoke Matlab commands, and a backend server that runs inside the internal Matlab JVM. The frontend and the backend are communicating over a socket connection, which in addition enables both parts to be on different machines on the network. Figure 6.2 shows the basic architecture of the Matlab Connector implemented.

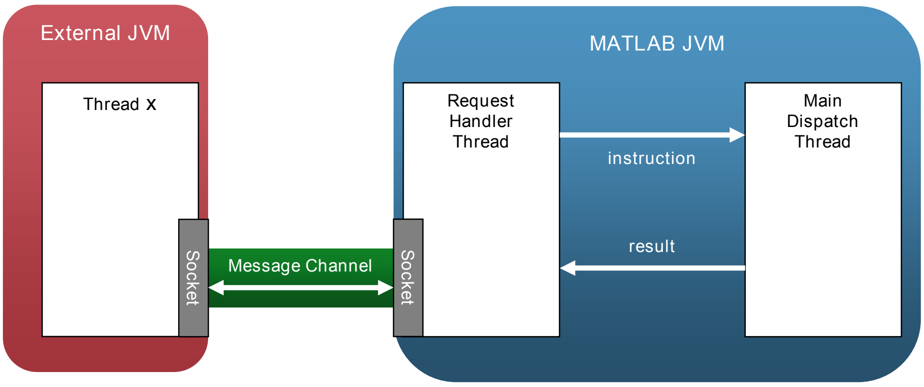


Figure 6.2: Schematic architecture of the Matlab connector.

The Matlab connector can be further used to extract implicit model properties directly from Simulink. A model could be loaded, compiled and simulated by calling Matlab commands from any external program, as long as it is connected to the Java Matlab backend.

Chapter 7

Results and Evaluation

In the following chapter a comparison of clone detection results between the proposed algorithm and an existing solution is made. The previously presented implementation is used to detect clones in a number of Simulink models.

7.1 Definition of Characteristic Properties

To be able to compare the results of different clone detection algorithms, key features of the result have to be defined. The result characteristic used in this chapter are the same as defined in chapter 6. These are also the features used in other publications:

# CG: The number of clonegroups found.

Avg. CG: The average number of fragments in a clonegroup.

# F: The overall number of cloned fragments found.

Avg. F: The average size of all detected fragments in the result set.

As additional to completeness and result relevance the scalability of a clone detection algorithm is a requirement, an algorithms running time is also of interest. But as the proposed clone detection tool is a rather prototypic implementation not focused on performance and further the running time behavior is strongly depending on an implementations architecture and platform, the measured numbers can only be considered as rough estimates.

7.2 Evaluation Environment

For comparing and evaluating the proposed clone detection algorithm, a set of models was selected. The set contains seven models that range from a very small model created explicitly for testing an algorithm for one contained clone to medium to large models from the automotive industry. These models are taken of current development project from industry partners and reflect therefore perfectly the kind of models that are subjects to clone detection algorithms in the industry.

Table 7.1 displays the models used. They range from a size of just a few blocks up to the size of nearly 30K blocks. When looking at the models it is also shown that the ratio from blocks to lines is very stable. By having almost the same amount of lines than blocks, it is clarified that the graphs representing these models are very sparse.

Model Name	Subsystems	Depth	Blocks	Lines	Ports
Model_A	2	1	26	27	50
Model_B	9	3	170	195	351
Model_C	80	9	1809	1531	3025
Model_D	199	7	2345	2701	5066
Model_E	204	7	3197	3474	6461
Model_F	1967	12	18041	18382	34305
Model_G	1604	7	27077	30718	55701

Table 7.1: An overview over the models used for benchmarking.

All tools and benchmarks presented in this chapter are run on a notebook that is based on an Intel Core I7 M620 running at 2.67GHz. The machine is equipped with 3GB of memory and with a conventional hard-drive. The operating system is Windows XP SP3 32bit running Java 1.6.

7.3 Comparison of Algorithms

To evaluate the clone detection results produced by the proposed clone detection algorithm, the results are compared to the results of ConQAT's algorithm. The new algorithm (NCD) is run in the presented implementation. ConQAT's algorithm is run in the ConQAT tool.

7.3.1 Completeness and Relevance

Table 7.2 shows the clone detection results of both algorithms. For ConQAT's algorithm, the numbers for Model_F and Model_G are missing. This is due to the fact that ConQAT was not able to process these models.

Model	**NCD**				**ConQAT**			
	# CG	**# F**	**Avg CG**	**Avg F**	**# CG**	**# F**	**Avg CG**	**Avg F**
Model_A	1	2	2	5	0	0	0	0
Model_B	1	2	2	8	1	2	2	6
Model_C	26	70	2,7	10,1	8	31	3,9	7,4
Model_D	26	98	3,8	8,8	26	80	3	8
Model_E	45	152	3,4	12,2	32	103	3,2	11,1
Model_F	37	950	25,7	10	-	-	-	-
Model_G	84	2313	27,5	13,5	-	-	-	-

Table 7.2: Comparison of clone detection results between the proposed tool and ConQAT. (# CG: Number of clonegroups, # F: number of fragments, Avg CG: average clonegroup size, Avg F: average fragment size).

It can be seen that the new algorithm yields a higher detection rate. This is not only true for the number of clonegroups that were detected, but it is especially true for the size of the detected fragments. The new solutions tends to detect larger fragments. This is not surprising, since the algorithm starts by creating as large as possible clone candidates.

As stated earlier, one of the big problems in model clone detection is to decide if a detected clone is relevant to a developer or for a given metric calculation. The results shown in table 7.2 were verified by sampling threw them with an experienced Simulink developer. For both algorithm only a few cases were found, that were not relevant at all. This can be taken as an indication that the additional clonegroups found by the new algorithm are not just multiples of already detected clones.

7.3.2 Running Time Behavior

The number in table 7.3 illustrate the running time behavior of both compared tools. The table shows the running time of the two algorithms in regard to the size of the input models. The running times in the table are the only the times taken by the clone detection algorithms. The time for parsing and preprocessing the models is not included in the displayed numbers.

Model	Blocks in Model	Runtime NCD [ms]	Runtime ConQAT [ms]
Model_A	26	0	94
Model_B	170	0	250
Model_C	1809	110	640
Model_D	2345	312	1797
Model_E	3197	422	1297
Model_F	18041	6300	-
Model_G	27077	35400	-

Table 7.3: Comparison of clone detection results between the proposed tool and ConQAT. (# CG: Number of clonegroups, # F: number of fragments, Avg CG: average clonegroup size, Avg F: average fragment size).

As stated before these numbers have to be taken as very rough estimates. It is no possible to conclude from the shown numbers directly to the running time behavior of the underlying algorithms. The algorithms are running in different tools that use very different data-structures and graph representations.

It can be seen from the numbers in table 7.3 that the proposed algorithm performs a little faster for each model than ConQAT's algorithm. But the even more interesting fact is that the new algorithm performs quite well even for the larger models. Model_G with more than 35K blocks is handled in about 35 seconds which is a time that that a developer can get over waiting for results. For small models with up to 5K blocks, the running times are so short, that the proposed algorithm can be seen as handling them in real-time.

7.4 Algorithm Limitations

A key weakness of the proposed algorithm is the detection of cloned cores from two given candidate fragments. The currently implemented method is rather rudimentary and leads to the loss of accuracy during clone detection. But the results show that even the simple matching implementation results in acceptable results.

By looking at the measured running times for the proposed algorithm, it can be seen clearly that the algorithm behaves not nearly linear with increasing model sizes. As the running times for the tested models were acceptable, it can be seen that the algorithm's implementation will not scale with even larger input models. It is to be doubted that the scalability can be significantly improved by optimizing the implementation details.

Chapter 8

Conclusions

8.1 Summary

In this thesis a solution for clone detection in Matlab Simulink models is presented. The main contribution of the solution is a clone detection tool build around a novel clone detection algorithm that is designed by analyzing the properties and structure of Simulink models. The new algorithm follows an approach to find as large as possible clone candidates. From these candidates sets of cloned fragments are extracted by finding the candidate's isomorphic cores.

Further the proposed tool provides the functionality of a complete clone detection pipeline including parsing and preprocessing Simulink models and presentation of the clone detection results.

8.2 Future Work

The presented algorithm's performance could be further improved by further investigating two parts of the clone detection pipeline. At first the algorithm could provide more significant results when more preprocessing effort is undertaken. This starts by further preprocessing of subsystems to a special handling of goto/from blocks. Secondly the algorithm for finding fragment's isomorphic cores is implemented in a very naive way. By re-using parts of the algorithms presented in 2.3 the detection rate can be further improved.

A further optimization for the algorithm could be the improvement of the pre-processing of the model. By including the rules of semantic normalization as presented in [1], the problem of semantic similar but structural different fragments can be solved. The approach presented by Al-Batran et al. has the disadvantage of relying on manual defined normalization rules. A further effort should be made here to either build a large directory of rules or to investigate possibilities of automated rule generation.

Optimizing clone detection running time performance is an other large field for improvements. As most implementations and tools available have a prototypic character, they are mostly not not optimized. By investigating the performance of used data-structures and by analyzing and finding implementation's bottle necks it should be possible to drastically improve an implementations performance.

Also a field that has not been investigated is the parallelization of clone detection algorithms. As multi-core processor systems become standard even in low cost desktop computers it seems natural to investigate how algorithms could be executed in parallel.

Bibliography

[1] B. Al-Batran, B. Schätz, and B. Hummel. Semantic clone detection for model-based development of embedded systems. In *MoDELS*, pages 258–272, 2011.

[2] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. Model clone detection in practice. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 57–64, New York, NY, USA, 2010. ACM.

[3] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 603–612, New York, NY, USA, 2008. ACM.

[4] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J.-F. Girard. An activity-based quality model for maintainability. In *In proc. 23RD International Conference on Software Maintenance(ICSM '07). IEEE Computer.* Society Press, 2007.

[5] dSPACE Inc. TargetLink Product Website. http://www.dspaceinc.com/en/inc/home/products/sw/pcgs/targetli.cfm. Visited 10/01/2012.

[6] Eclipse Foundation, Inc. Eclipse Rich Client Platform. http://www.eclipse.org/home/categories/rcp.php. Visited 18/01/2012.

[7] S. Fortin. The graph isomorphism problem. *Journal of Computational Chemistry*, 12(10):1–5, 1996.

[8] M. Garey and D. Johnson. *Computers and Intractability.* Freeman and Co., New York, 1979.

[9] N. Gold, J. Krinke, M. Harman, and D. Binkley. Issues in clone classification for dataflow languages. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 83–84, New York, NY, USA, 2010. ACM.

[10] hello2morrow Inc. Sotograph Product Website. http://www.hello2morrow.com/products/sotograph. Visited 05/01/2012.

[11] W. Holoch. Metriken zur qualitätsmessung von simulink-modellen in der modellgetriebenen softwareentwicklung. Master's thesis, Ulm University, 2011.

[12] B. Hummel, E. Juergens, and D. Steidl. Index-based model clone detection. In *Proceeding of the 5th international workshop on Software clones*, IWSC '11, pages 21–27, New York, NY, USA, 2011. ACM.

[13] Java Universal Network/Graph Framework. Project Website. http://jung.sourceforge.net. Visited 20/01/2012.

[14] E. Juergens, F. Deissenboeck, and B. Hummel. Clonedetective - a workbench for clone detection research. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 603–606, Washington, DC, USA, 2009. IEEE Computer Society.

[15] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *in Proc. ALENEX 2007. SIAM, Philadelphia*, 2007.

[16] T. Junttila and P. Kaski. Conflict propagation and component recursion for canonical labeling. In A. Marchetti-Spaccamela and M. Segal, editors, *Theory and Practice of Algorithms in (Computer) Systems*, volume 6595 of *Lecture Notes in Computer Science*, pages 151–162. Springer Berlin / Heidelberg, 2011.

[17] R. Koschke, E. Merlo, and A. Walenstein, editors. *Duplication, Redundancy, and Similarity in Software, 23.07. - 26.07.2006*, volume 06301 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[18] L. M. Laird and M. C. Brennan. *Software Measurement and Estimation: A Practical Approach*. IEEE Computer Society, Washington, DC, USA, 2007.

[19] B. T. Messmer and H. Bunke. Subgraph isomorphism detection in polynominal time. In *ACCV*, pages 373–382, 1995.

[20] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering*, volume 5503 of *Lecture Notes in Computer Science*, pages 440–455. Springer Berlin / Heidelberg, 2009.

[21] Phaedrus Systems Ltd. QAC Product Website. http://www.phaedsys.org/principals/programmingresearch/pr-qac.html. Visited 05/01/2012.

[22] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 276–286, Washington, DC, USA, 2009. IEEE Computer Society.

[23] E. Pretschner, M. Broy, I. H. Krüger, and T. Stauner. Software engineering for automotive systems: A roadmap. In *in Future of Software Engineering (FOSE '07)*, pages 55–71, 2007.

[24] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *School of Computing TR 2007-541, Queen's University*, 115, 2007.

[25] J. Scheible. Ein framework zur automatisierten ermittlung der modellqualität bei eingebetteten systemen. In *MBEES*, pages 1–6, 2010.

[26] Technical University Munich. Continuous Quality Assessment Toolkit Website. http://www.conqat.org. Visited 12/01/2012.

[27] The MathWorks Inc. MATLAB Product Website. http://www.mathworks.com/products/matlab/. Visited 06/01/2012.

[28] The MathWorks Inc. Polyspace Product Website. http://www.mathworks.com/products/polyspace/. Visited 05/01/2012.

[29] The MathWorks Inc. Simulink Coder Product Website. http://www.mathworks.com/products/simulink-coder. visited 10/01/2012.

[30] The MathWorks Inc. Simulink Product Website. http://www.mathworks.com/products/simulink/. Visited 06/01/2012.

[31] The MathWorks Inc. The Mathworks Automotive Advisory Board. http://www.mathworks.com/automotive/standards/maab.html. visited 19/01/2012.

[32] The Motor Industry Software Reliability Association. MISRA Website. http://www.misra.org.uk. Visited 10/01/2012.

[33] D. Travkin and I. Stürmer. Tool supported quality assessment and improvement in matlab simulink and stateflow models. In *Postproceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4)*, Paderborn, Germany, 2008.

[34] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1):31–42, Jan. 1976.