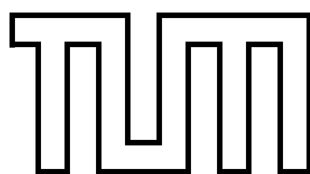


FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl IV
Software & Systems Engineering

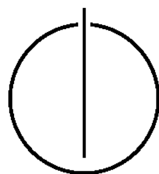
Master's Thesis in Informatics

Improving clone detection for models

Verbesserung der Klonerkennung für Modelle

Author: Michael Pfähler
Supervisor: Prof. Dr. Dr. h.c. Manfred Broy
Advisor: Benjamin Hummel

Date: November 15th 2009



I assure the single handed composition of this master's thesis only supported by the declared resources.

München, November 15th 2009 Michael Pfähler

Acknowledgments

I want to express my utmost gratitude to my advisor Benjamin Hummel for his guidance and support during the creation of this work. Working with you was always fruitful and pleasantly uncomplicated.

I am very thankful for the discussions with Dr. Bernhard Schätz. Many of the ideas for this work evolved during our talks. Thank you for your kindness and deep insights in modeling with Matlab/Simulink.

Special thanks go to Christian Kühnel. Thank you for your input and support in assessing the practical suitability of our solutions. I hope, the results are useful for your future daily work. Thank you and Andreas Siller for taking the time to participate in our study.

Thanks to Florian Deißenböck and Elmar Jürgens for answering my technical questions and your introduction to clone detection.

Nguyen Anh Hoan, thank you for your detailed and honest answers to my questions regarding the algorithms developed by your research group.

Finally, thanks to all proof-readers, my LaTeX-speaking lab mates, coffee and lunch companions that made the last six months a wonderful time.

Thank you.

Abstract

Within the last years, the importance of model-driven development to describe complex systems has continuously increased. Due to their size, models in the area of automotive software engineering require growing maintenance efforts. Just as in code-based development, the maintainability of a model is adversely affected by the existence of duplicate elements (*clones*) within a model. Available tools for clone detection in data flow graphs currently possess a weakness: Developers consider many of the reported clones as irrelevant.

In this thesis, we present an overview of different algorithms for clone detection within data flow graphs and evaluate the approaches *w.r.t.* their practical suitability for large-scale MATLAB/Simulink models. We present a new technique to detect clones on the level of Simulink subsystems. In order to better assess the relevance of reported clones, we examine a number of clone metrics which have not been considered up to now. We evaluate the suitability of the identified clone metrics in a case study with an automotive engineering partner.

Contents

Acknowledgements vii

Abstract ix

1. Improving clone detection for models 1

2. Basic principles 3
- 2.1. Graph theoretic definitions . . . 3
- 2.2. MATLAB/Simulink . . . 5
- 2.3. Clone detection . . . 6
  - 2.3.1. Model clone detection in data flow graphs . . . 7
  - 2.3.2. Subsumption of clone detection within graph mining . . . 8
  - 2.3.3. Complexity of clone detection . . . 9

3. Algorithms for graph-based model clone detection 11
- 3.1. Preparatory steps . . . 11
- 3.2. ModelCD . . . 11
  - 3.2.1. Fundamentals . . . 12
  - 3.2.2. *eScan* . . . 13
  - 3.2.3. *aScan* . . . 19
  - 3.2.4. ModelCD Optimizations . . . 20
- 3.3. ConQAT's Model Clone Detection . . . 21
- 3.4. *gSpan* . . . 22

4. Improving clone relevance 23
- 4.1. Factors affecting clone relevance . . . 24
- 4.2. Assessing clone relevance in CMCD . . . 24
- 4.3. Graph distance filters . . . 25
  - 4.3.1. Diameter filter . . . 25
  - 4.3.2. Average distance filter . . . 25
  - 4.3.3. Zero-weight filter . . . 26
- 4.4. Clone ranking scheme . . . 26
  - 4.4.1. Interface ranking value . . . 26
  - 4.4.2. Relative weight ranking value . . . 27
- 4.5. Ranking scheme evaluation . . . 27

5. Evaluation of model clone detection algorithms 29
- 5.1. Models used for algorithm evaluation . . . 29
- 5.2. ModelCD . . . 30

5.2.1. ModelCD Optimizations 30
5.2.2. *eScan* 33
5.2.3. *aScan* 36
5.2.4. Clone grouping in ModelCD: Completeness vs. redundancy 36
5.3. Cloned subsystem detection 37
5.4. ConQAT's Model Clone Detection 38
5.5. *gSpan* 40
5.6. Runtime overview 40
5.7. Summary 41

6. Evaluation of clone relevance improvements 43
6.1. Graph distance filters 43
6.2. Clone ranking scheme 44

7. Conclusion and future work 47
7.1. Conclusion 47
7.2. Future work 48

Appendix 53

A. Detailed case study results 53

Bibliography 55

1. Improving clone detection for models

Within the last years, model-driven development has become a popular paradigm to create software systems. Domain-specific models describe systems with concepts that are meaningful to an application domain expert. An automated process transforms models into executable software.

In the automotive domain, MATLAB/Simulink[1] is widely used to model embedded systems. Developers are able to describe complex computations with a set of function blocks and a data flow between those blocks.

In traditional, code-based software development, the reuse of code segments by copy & paste, called *cloning*, has negative effects on maintainability [2]: Expressing the same functionality at multiple locations within the code base can lead to inconsistent changes and faults.

Cloning also occurs in model-driven development when a developer copies model elements instead of using an appropriate reuse mechanism. Although no studies have been performed that explicitly examine the effects of cloning in model-driven development, cloning is likely to also hamper the maintainability of models.

The detection of model clones has been an active area of research within the last years. There exist several approaches to detect clones in models [3, 4, 5]. However, these techniques possess a number of shortcomings: One strategy uses a heuristic to detect clones and is thus potentially incomplete. Other approaches have not yet been thoroughly evaluated *w.r.t.* their suitability for large models encountered in practice. Furthermore, the techniques focus solely on a mathematical definition of clones and do not well consider a clone's relevance to developers.

In this work, we evaluate existing approaches for model clone detection in data flow graphs. The analysis focuses primarily on ModelCD [5], the most recently proposed technique for clone detection in graph-based models. We evaluate its suitability for large-scale models.

Furthermore, we study a number of measures to increase the relevance of the identified clones. We propose an algorithm to accurately detect highly-relevant clones, study the suitability of clone filters *w.r.t.* specific graph properties and evaluate a ranking scheme for clones based on a variety of metrics.

The thesis is structured in seven chapters. Chapter two provides an overview of graph theoretic foundations, a short introduction into MATLAB/Simulink, and a panoramic view of clone detection in code and data flow graphs. Chapter three details the functionality of a number of algorithms that can be applied to identify clones within a model. Chapter four mentions factors influencing the relevance of clones to developers and proposes measures to better assess a clone's relevance. Chapter five contains the experiences we gained during the evaluation of the algorithms introduced in chapter 3 and proposes possible improvements. Chapter six contains our findings *w.r.t.* the clone relevance improvements detailed in chapter five. Chapter seven concludes our findings and gives an outlook on future work.

2. Basic principles

This chapter provides an overview of graph theoretic foundations, gives a short introduction into MATLAB/Simulink and defines the clone detection problem for code and graph-based models. We relate model clone detection to graph mining and consider the algorithmic complexity of clone detection in graph-based models.

2.1. Graph theoretic definitions

Graph theory is an area of mathematics that uses graphs as a representation of relational data. The abstraction of a graph serves as a foundation for many problem formulations.

A graph $G = (V, E)$ consists of a set of nodes (also called vertices) V and a set E of pairs $\{u, v\}$, $u, v \in V$, called edges.

An edge $e = \{u, v\}$ is said to be *incident* to the vertices u and v it connects.

A graph is usually visualized by drawing a dot for each vertex and a line between two vertices v_1 and v_2 if there exists an edge $\{v_1, v_2\} \in E$. We only care about the nodes' topology and not about a specific layout.

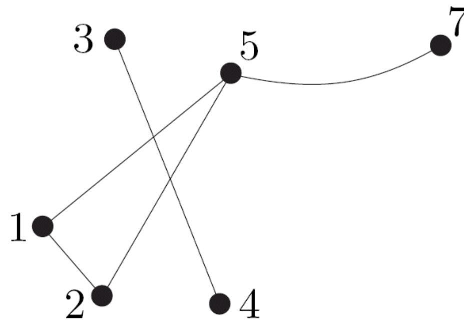


Figure 2.1.: A simple graph

A *directed* graph (called digraph) expresses connectivity information with a set of ordered pairs (u, v) called *arcs* or *directed edges*. An arc (u, v) is said to be directed from u to v. To disambiguate that the edges of a graph are *not* directed, a graph can be explicitly qualified as an *undirected* graph.

A *labeled* graph uses a labeling function $L : V \cup E \rightarrow N$ to assign a label from some set N to its nodes and edges.

The *degree* $d(v)$ of a node v is the number of edges incident to v. For digraphs, a node's degree can further be subdivided into the number of incoming edges (*in-degree*) and the number of outgoing edges (*out-degree*).

In *complete* graphs, each vertex is connected to every other vertex, *i.e.*, $E = V \times V$.

A *multigraph* can contain multiple edges between two pairs of nodes. The edges of a multigraph are therefore defined as a multiset.

A *path* in a graph is a sequence of edges E such that the target node of each edge in the sequence is the source node of the following edge.

A graph is *(weakly) connected* if there exists a (undirected) path from each vertex to any other vertex in the graph. The graph depicted in Figure 2.1 is not connected as, *e.g.*, there exists no path from node 4 to node 5.

A graph $G' = (V', E')$ is a *subgraph* of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. If G' is a subgraph of G we say that G *contains* G' (written as $G' \subseteq G$). If $G = (V, E)$ contains $G' = (V', E')$ and E' contains all edges $\{x, y\} \in E$ with $x, y \in V'$ then G' is an *induced subgraph* of G.

A subgraph G' of G that is complete is called a *clique*. A *maximal clique* is a set of vertices that induces a complete subgraph, which is not a subset of the vertices of any larger complete subgraph.

An *independent set* is a set of vertices of which no two nodes are adjacent in the graph. A *maximal independent* set is an independent set that is not a subset of any other independent set.

The *diameter* of a graph is the length of the longest shortest path in a graph. To determine a graph's diameter one calculates the lengths of all shortest paths between each pair of nodes. The greatest length of any of these paths is the diameter of the graph. The *eccentricity* of a node v is the maximum shortest distance from v to any other node in the graph. Thus a graph's diameter is the maximum eccentricity of all nodes within a graph

Graph isomorphism Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if they are topologically identical, *i.e.*, there exists a bijection $f : V_1 \to V_2$ with $\{x, y\} \in E_1 \Leftrightarrow \{f(x), f(y)\} \in E_2$. In other words, there exists a mapping from V_1 to V_2 such that each edge in E_1 is mapped to an edge in E_2 and vice versa. For labeled graphs, the mapping must also preserve the edge and node labels.

Determining if two graphs are isomorphic is computationally very demanding as, in general, there exists no order on nodes and edges. It is unknown whether the determination of graph isomorphism lies in P or is NP-complete [6].

Certain graph properties remain *invariant w.r.t.* to isomorphic graphs. For example, a graph's ordered degree sequence is shared by all isomorphic graphs. The equality of the degree sequence is a necessary condition for isomorphism but not a sufficient one: To prove that two graphs are actually isomorphic requires a more advanced technique such as canonical labeling.

Canonical labels A *canonical label* is a unique graph code that is invariant on a particular ordering of a graph's nodes and edges. Two graphs have the same canonical label if and only if they are isomorphic. To compute and compare two canonical labels is as complex as to determine if two graphs are isomorphic. Computing a canonical label can therefore be expensive. However, there exist ways to reduce the computational complexity. We will detail one approach in Section 3.2.2.3. An advantage of using canonical labels is that, once a

canonical label has been computed, further comparisons with other canonical labels merely require the comparison of two strings.

This work contains only those aspects of graph theory that are relevant for graph-based model clone detection. Further information and definitions about graph theory and its applications can be found, *e.g.*, in [7].

2.2. MATLAB/Simulink

MATLAB is a software package provided by *The MathWorks Inc.* for numerical mathematics that enables its users to easily perform vector and matrix calculations. MATLAB consists of a base module that offers many predefined mathematical functions, visualizations and interfaces to other programming languages and hardware. MATLAB can be extended by several toolboxes, *e.g.*, for control systems, signal processing and optimization.

Simulink is a toolbox that offers a graphical user interface for the modeling, simulation and analysis of dynamic systems via data flow graphs. Simulink offers an extensive library of predefined function and parameter blocks for linear, non-linear, discrete and hybrid systems. Data flow graphs consist of instances of function blocks that are connected to each other by signal flow lines. A block instance can be associated with a set of attributes depending on the block's type.

Most Simulink blocks have one or more incoming and outgoing ports to which lines can be connected. A user can assign a custom name to each block requiring that the name is unique within a defined scope. In order to better understand Simulink we will explain two exemplary models.

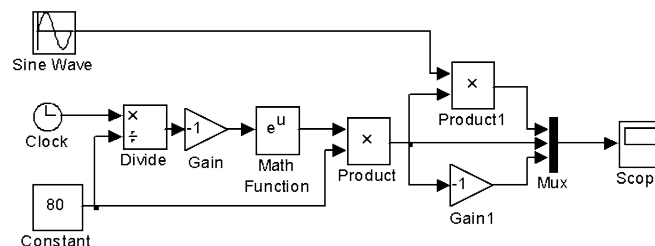
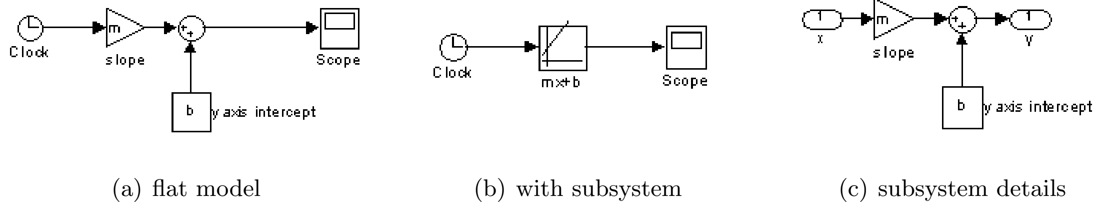


Figure 2.2.: Simulink model for $f(t) = 80 \cdot \exp(-\frac{1}{80}t) \cdot \sin(0.25t + \frac{\pi}{3})$

The Simulink model in Figure 2.2 represents a mathematical function. t stands for the simulation time which is provided by the *Clock* block in the model. The *Constant* block emits a time independent scalar value of 80. The Clock and Constant values are input to a *Divide* block that calculates the fraction $\frac{t}{80}$. The following *Gain* block multiplies this fraction with -1. The output of the Gain block serves as input to a *Math Function* block which applies an exponential function to its input value. The *Product* block multiplies the result of the Math Function block with the Constant block's value of 80. The result of the Product block is multiplied with the result of the *Sine Wave* block modeling a sine wave with a frequency of 0.25 and a phase of $\frac{\pi}{3}$ by the block with the label 'Product1'. The result of this multiplication, a damped sinus oscillation together with its envelopes, is transferred to a *Scope* block. When simulating the model, the user can see the output

on this oscilloscope. The *Mux* block combines three scalar values into a composite vector signal to achieve a clearer arrangement of lines. A composite signal can be compared to a cable bundle.



(a) flat model (b) with subsystem (c) subsystem details

Figure 2.3.: Creating a subsystem

Users can combine sets of blocks and lines into subsystems and thus create higher-level domain abstractions like, *e.g.*, a PID controller. The use of subsystems enables the hierarchical decomposition of a Simulink model. If a subsystem shall be used at multiple locations, it should be externalized as a library element. To use the functionality provided by a library element, a model can reference a library element together with a set of input parameters. Apart from providing custom parameters, a library element can not be adapted to specific requirements that the calling model element might have.

Figure 2.3 displays the process of creating a subsystem: Figure 2.3(a) displays a flat model that describes the calculation of a straight line with the equation $y = mx + t$. The input parameter x is bound to the simulation time. The calculation of straight lines can be seen as a domain concept and it makes sense to create a subsystem that calculates the line equation for an input parameter. Figure 2.3(b) shows how the resulting subsystem *mx+b* can then be used just as any other block in the Simulink library. Figure 2.3(c) shows the contents of the newly created subsystem: The subsystem receives its input values via the in-port x that routes the signal that is connected to the subsystem's input port in Figure 2.3(b). The subsystem's out-port y is accordingly wired to the subsystem's output port. The newly created subsystem *mx+b* can now be externalized into a library element. The subsystem can then be referenced from all model parts that require the calculation of a straight line.

Instead of referencing a subsystem in a library, developers sometimes copy parts of a model to another location and adapt it to their specific needs. This leads to redundancy in the model and can cause maintenance problems as it is discussed in later sections of this work.

A more detailed explanation on MATLAB and Simulink can be found in [1].

2.3. Clone detection

The Merriam-Webster dictionary defines a *clone* as *one that appears to be a copy of an original form*, thus being synonymous to a *duplicate*. Clone detection is the process of identifying cloned entities. In software engineering, a lot of research has been conducted on the identification of cloned program code fragments [8, 9, 2, 10]. Having the same code at different locations is contradictory to the "don't repeat yourself"-principle taught in

beginner's programming courses and is regarded as the number one bad software practice in Kent Beck's and Martin Fowler's collection of "code smells" [11]. Studies have shown that clones increase maintenance costs and may lead to faults when changes to clones are not performed consistently [2].

However, in the software engineering field there is no common, clear-cut definition of what a software clone exactly is. According to Koschke [9], the most agreed-upon, yet vague definition of software code clones is that clones are *segments of code that are similar according to some definition of similarity*. As a consequence, researchers usually define their understanding of similarity up-front.

For the remaining parts of this thesis we will say that two segments of code are *in a cloning relationship* if they are similar *w.r.t.* the similarity measure used. We term a segment of code that is in a cloning relationship with another segment a *clone instance*. A *clone group* contains two or more clone instances. The cloning relationship relation is usually transitive, *i.e.*, all clone instances in a clone group are in a cloning relationship with all other clone instances within the group.

In [9], Koschke identifies three different ways in which code fragments can be similar to each other: The similarity can be based on text, syntactic structure or semantics.

Text-based similarity is defined through the exact same textual representation of both clone instances. For syntactic similarity, some of the details of a merely textual representation are abstracted away, such as a variable's concrete identifier. Semantic similarity requires clone instances to have a common observable behavior. In other words, semantic clones have identical post-conditions for equal pre-conditions. Unfortunately, semantic similarity is undecidable in general.

In [12], Kim *et al.* adduce time pressure, programming language limitations, educational deficiencies, and other organizational issues as reasons for the introduction of clones into software systems.

A number of tools have been developed to identify cloned fragments in program code [13, 10]. These tools help developers to raise the awareness of clones existing within their systems and aim to prevent the creation of new clones.

2.3.1. Model clone detection in data flow graphs

Within the last years, model-driven software development (MDSD) has become a popular way of building software systems. Developers can define software systems on a higher level of abstraction, using concepts that are meaningful to an application domain expert. As models are used to generate code, they can be regarded as a higher level programming language. Since most of the reasons leading to clones in code-based development are also valid for MDSD, it is not surprising that clones can also be found in models.

For our studies of model clone detection, we focused on data flow graphs. Our experiments were performed using MATLAB/Simulink models.

In graph-based clone detection, a clone instance corresponds to a subgraph G' of the model graph G. Just like for code-based clone detection, there exist multiple ways to define the similarity between two subgraphs:

Semantic similarity means that two subgraphs model the same concept in the application domain. It would be an ideal goal to identify all semantic clones within a model. Deciding if two subgraphs model the same concept is often not trivial. Most clone detection techniques

therefore use methods to approximate semantic similarity using graph isomorphism and distinguish *exact* and *approximate* clone detection.

For exact clone detection, a clone instance is defined as a weakly connected subgraph $G_1 = (V_1, E_1)$ of G that is isomorphic to at least one other subgraph $G_2 = (V_2, E_2)$ of G *w.r.t.* the labeling function L.

For approximate clone detection, two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ of G are considered similar if they contain two "relatively big" label isomorphic subgraphs $G'_1 = (V'_1, E'_1) \subset G_1$ and $G'_2 = (V'_2, E'_2) \subset G_2$. Approximate clone detection thus subsumes exact clone detection, as approximate clone detection also identifies all exact clones if $G'_1 = G_1$ and $G'_2 = G_2$.

The labeling function L greatly influences which subgraphs are considered isomorphic. The assignment of labels to nodes and edges will be referred to as the *normalization* of a model. Normalization labels should contain only relevant information about nodes and edges, *s.t.* model elements with the same normalization label can be considered duplicates of each other. Detecting model clones on a normalized model graph can be compared to syntactic code clone detection in that only important information is considered.

An interesting criterion for model clone detection is whether cloned subgraphs are allowed to overlap. Most algorithms we studied do not allow clones to overlap, *i.e.*, $V_1 \cap V_2 = \emptyset$ for two cloned subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. However, there may exist cases where overlapping subgraphs are interesting.

2.3.2. Subsumption of clone detection within graph mining

Graph-based clone detection can be regarded as a specialization of *graph mining*. Graph mining deals with the extraction of interesting structures from graphs. Its purpose, the retrieval of useful knowledge, is therefore similar to the aims associated with data mining. However, in contrast to data mining, graph mining works on structured data represented as a graph.

One common use case, called *graph matching*, aims at the identification of frequent patterns in a graph. A frequent pattern is defined as a subgraph G with a number of label isomorphic copies, called *embeddings* of G.

Commonly, a set of graphs serves as input to the graph matching problem. The number of input graphs that contain a subgraph G is termed the *support* of G. In this setting, a subgraph is considered *frequent* if its support is above a defined threshold.

If a single graph serves as input to the graph matching problem, we call this a *single graph setting*. In this setting, a subgraph G is considered *frequent* if it occurs at least a defined number of times within the input graph. This minimum required pattern frequency is configured by an input parameter.

Graph-based model clone detection therefore corresponds to a graph matching problem within a single graph and a minimum required pattern frequency of 2.

A number of complete graph mining algorithms have been developed that could be used for clone detection. However, these algorithms may not be appropriate for our purpose as they usually work with a higher required minimum pattern frequency.

More information on graph mining can be found in [14].

2.3.3. Complexity of clone detection

Clone detection in graph-based models is the problem of identifying all maximum isomorphic subgraphs within a graph. It is related to the NP-complete largest common subgraph-isomorphism (LCS) decision problem [15]. The LCS problem takes two graphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ and an integer k as an input and decides whether there exists a subgraph G of G_1 and G_2 with at least k edges. Clone detection differs from this problem in that we are looking for isomorphic subgraphs in only one graph and search for all duplicate maximum subgraphs. It is possible to reduce the LCS problem to the clone detection problem: The LCS problem can be decided by detecting all clones within a graph $G = (V_1 \cup V_2, E_1 \cup E_2)$ and deciding if the biggest clone contains at least k edges. The clone detection problem is therefore at least NP-complete. An optimal clone detection algorithm has exponential complexity in its worst case. However, as most Simulink models are sparse graphs, an algorithm may still process a model within reasonable time and space boundaries.

We will present an overview of existing algorithms for clone detection in graph-based models in the next chapter.

3. Algorithms for graph-based model clone detection

This chapter presents an overview of algorithms for graph-based model clone detection.

Section 3.1 explains how a Simulink model is converted into a normalized graph which contains only information that is relevant for clone detection.

Section 3.2 details the ModelCD algorithms *eScan* and *aScan*. In Section 3.2.2.4, we present vSIGRAM, a graph mining algorithm that inspired the creation of *eScan*.

We introduce ConQAT's Model Clone Detection algorithm in Section 3.3 and briefly describe the graph mining algorithm *gSpan* in Section 3.4.

3.1. Preparatory steps

As a first preliminary step, a graph representation of the model under analysis is created. For this purpose, a Simulink model is converted into a labeled, directed multi-graph $G = (V, E, L)$ in which the set of nodes V consists of the set of Simulink blocks and the signal flow lines between the Simulink blocks make up the graph's edge multiset E. E is a multiset as a Simulink model can contain more than one edge with the same source and target node.

As a next step, a labeling function $L : V \cup E \rightarrow N$ assigns normalization labels to nodes and edges. Each node is assigned a normalization label that is based on its block type and other decisive properties. For example, the label for a *RelationalOperator* node could also contain the concrete operation (*e.g.*, >=) it performs, in order to distinguish it from *RelationalOperator* blocks performing other operations.

The assignment of block labels is surjective, *i.e.*, the same label can be used for different block types. For example, it makes sense to use the same label for Divide and Product blocks, as a product could also be expressed as a division using the multiplicand's reciprocal value.

Edges can receive a label that helps to discriminate between their original source and target ports.

The normalization rules heavily impact the clone detection results, as they define which model elements can map to each other under the isomorphism relation.

3.2. ModelCD

ModelCD (Model Clone Detection) is a graph-based model clone detection tool, presented by Pham *et al.* in [5]. ModelCD consists of two algorithms for exact and approximate clone detection and a set of common optimization strategies. We will detail both algorithms and the optimizations in the following sections.

The basic idea used by ModelCD to detect clones is to identify bigger clones by adding edges to smaller, already detected clones.

3.2.1. Fundamentals

This subsection contains an overview of fundamental terminology and concepts specific to ModelCD.

Pham *et al.* term a weakly connected subgraph containing k edges a *fragment of size k* (f_k). A *cloned fragment* corresponds to a subgraph which is in a cloning relationship with another, non-overlapping subgraph. A clone group contains two or more fragments that are in a cloning relationship. For ModelCD, all clone instances within a clone group are not allowed to overlap with any other clone instance in the group.

The fundamental observation used by ModelCD, is that cloned fragments of size k ($k > 0$) in turn contain smaller, cloned subgraphs of size k-1. Vice versa, cloned fragments of size k can be created through the addition of equivalent edges to the corresponding cloned fragments of size (k-1). We refer to the inclusion of an additional edge e to a fragment f_k as the *extension operation* $\oplus$. This new edge e must not already be part of f_k, but at least one of the nodes that e connects has to be part of f_k, in order to yield a weakly connected subgraph with k+1 edges as result.

We use the term *offspring set of* f_k*, denoted by* $O(f_k)$*,* to describe a recursively defined set containing fragments that were created through an extension of f_k or through the extension of a fragment within $O(f_k)$.

A clone group G_1 is said to be *covered* by another group G_2 if all members of G_1 are a subgraph of at least one member of G_2. In Figure 3.1, geometric shapes denote clone groups. The clone group B, depicted as rectangles, is covered by clone group A, shown as ovals. However, clone group C, illustrated by pentagons, is not covered by A because one instance of C is not part of an instance of A.

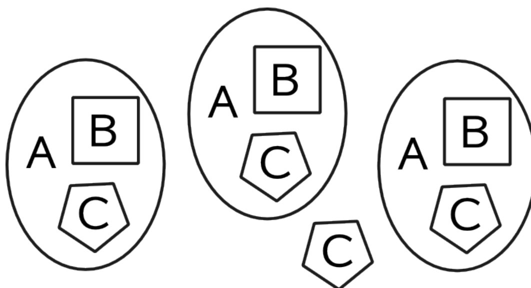


Figure 3.1.: Clone group B is *covered*

Conceptionally, ModelCD uses a layered graph data structure, called *"clone lattice"*, during clone detection. Layer k of this lattice contains all cloned fragments of size k as nodes. The clone lattice contains an edge from a fragment node on layer k to a fragment node on layer k+1 iff the cloned fragment on layer k+1 can be created by the addition of one edge to the cloned fragment on layer k. Clone detection corresponds to a traversal of this clone lattice.

3.2.2. eScan

eScan is a clone detection algorithm that identifies exact clones within a model graph. Two subgraphs are considered similar if they are isomorphic *w.r.t.* the labeling function in use. The way in which frequent subgraphs are identified is very similar to vSiGraM, a complete graph matching algorithm described in [16]. We will detail the differences to *eScan* later in this section.

The core *eScan* detection routine performs a depth first traversal of the clone lattice. *eScan* uses a generating parent technique to ensure that each fragment is processed only once.

```
1  function eScan(G = (V,E,T))
2    L_1 ← { all cloned 1−fragments }
3    for each f_1 ∈ L_1 do Discover(f_1, Clones(f_1))
4    for each L_k do CG ← CG ∪ Group(L_k)
5    Filter(CG)
6  return CG
7
8  function Discover(f_k, Clones(f_k))
9    for each g_k ∈ Clones(f_k) do
10     C_{k+1} ← C_{k+1} ∪ {g_k ⊕ e | e ∈ E}
11   for each c_{k+1} ∈ C_{k+1} do
12     if GeneratingParent(c_{k+1}) = f_k then
13       Find(Clones(c_{k+1}))
14       if |Clones(c_{k+1})| > 1 then
15         L_{k+1} ← L_{k+1} ∪ Clones(c_{k+1})
16         Discover(c_{k+1}, Clones(c_{k+1}))
```

Listing 3.1: Exact clone detection

3.2.2.1. The discovery algorithm

The *eScan* algorithm is depicted in Listing 3.1. At first, the set of all cloned 1-fragments, L_1, is determined (line 2). A 1-fragment is a subgraph of the model graph consisting of two nodes connected by one edge. In line 3, the function *Discover* is called for each cloned fragment f_1 together with a set of all fragments isomorphic to f_1 (including f_1). In lines 9-10, *Discover* builds a set of candidate fragments C_{k+1}. The candidate fragments of size k+1 are constructed by performing the extension operation on all fragments in $Clones(f_k)$. The extension of a fragment $c_k \in Clones(f_k)$ produces n candidate fragments where n is the number of edges adjacent to, but not contained in c_k. In lines 13-16, for each candidate fragment c_{k+1}, if it is a cloned fragment, c_{k+1} and its clones are added to L_{k+1} and used for a recursive call to *Discover*. The size of the fragment processed therefore grows by one with each recursive call. The recursion backtracks when no clones for fragments of size k+1 can be found. Identifying cloned fragments for a candidate fragment c_{k+1} is achieved by the comparison of the canonical label of c_{k+1} with all other fragments in C_{k+1}. If

for a candidate fragment c_{k+1} at least one other, non-overlapping fragment exists, c_{k+1} is considered a cloned fragment.

Generating parent strategy It is important to note that a candidate fragment c_{k+1} can be created by the extension of up to k+1 fragments of size k, depending on which of its edges is added last. In order to avoid redundant processing of fragments, *eScan* uses a *generating parent* technique which assigns for each candidate fragment c_{k+1} a unique fragment of size k that may be extended to yield c_{k+1}. Figure 3.2 shows an example in which 3 fragments of size 2 could be used to create c_{k+1}. However, only one of the smaller fragments, f_k, is allowed to create c_{k+1}. Thus, f_k is the generating parent of c_{k+1}

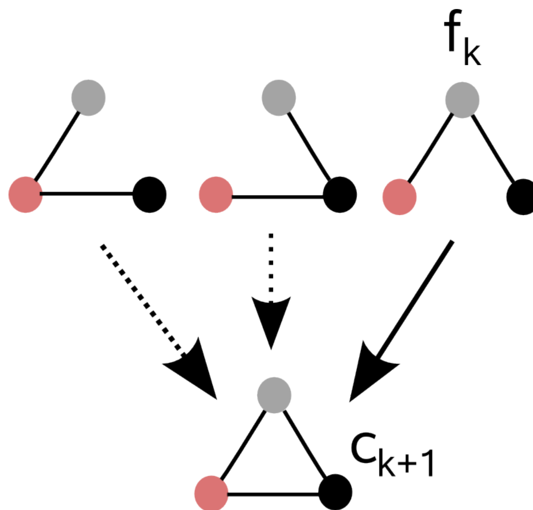


Figure 3.2.: f_k is the generating parent of c_{k+1}

This techniques ensures that each fragment is processed exactly once. The identification of a candidate fragment's generating parent is based on a unique order of edges given by the fragment's canonical label. f_k is the generating parent of c_{k+1} if and only if the last edge in the unique order that does not disconnect c_{k+1} is the edge that was used for the extension of f_k which resulted in c_{k+1}. In line 12, the DFS continues only for those candidate fragments for which f_k is the generating parent.

3.2.2.2. Clone grouping and filtering

After the DFS terminates, all cloned fragments are contained within the clone lattice. In line 4 of Listing 3.1, all cloned fragments are organized into clone groups based on their canonical labels.

At this point, those clone groups consisting of more than two clones may contain overlapping fragments. ModelCD does not allow overlapping fragments within clone groups. The *filter* function identifies and splits these groups into a set of smaller, non-overlapping, maximal clone groups. For an example, let us assume clone detection identified a clone group S consisting of four fragments (a,b,c,d) that are all isomorphic to each other. However, c overlaps with b and d. According to the clone definition used by ModelCD, S is not a

clone group. To overcome this shortcoming, *filter* builds an overlap graph with the cloned fragments as nodes and an edge between them if both fragments have no common nodes. For our example, the graph $G = (\{a, b, c, d\}, \{\{a, b\}\{a, c\}\{b, d\}\{a, d\}\})$, depicted in Figure 3.3 is built. *eScan* then applies a maximal clique cover algorithm that returns two clone groups: (a,b,d) and (a,c).

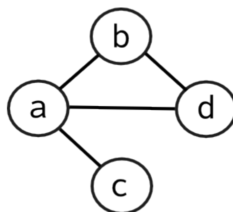


Figure 3.3.: Overlap graph for clone group S

After this step, CG contains only non-overlapping clone groups. However, there may still exist covered groups. Therefore, as a last step, all covered groups are removed from CG.

3.2.2.3. Computing canonical labels

As mentioned in Section 2.1, a canonical label is a unique graph code that is invariant on a particular ordering of the graph's nodes and edges. Isomorphic graphs share a common canonical label. *eScan* uses an approach described in [17] to build canonical labels.

In general, it is expensive to determine if two graphs are isomorphic, as there is no defined order on nodes and edges. On the other hand, it is easy to check whether two graphs G_1 and G_2 are isomorphic if they have unique node labels: One can use the order defined by the node labels to create a unique adjacency matrix for G_1 and G_2. One can then simply check if the adjacency matrices are equal. This check can be performed in $O(|V_1|^2)$ by comparing all entries of the adjacency matrices.

For two unlabeled graphs U_1 and U_2, the situation is harder as there exist multiple adjacency matrices. To determine if U_1 and U_2 are isomorphic, one could first pick a random adjacency matrix for U_1 and then test all potential adjacency matrices of U_2 for equality with the adjacency matrix of U_1. The complexity of all equality checks is $O(|V_1|! \cdot |V_1|^2)$ - making this approach impractical for even a relatively small number of nodes.

This brute force technique of enumerating all adjacency matrices can be improved by classifying vertices: One could for example partition all nodes into subsets V_i, depending on their degree d. Any node with degree d in U_1 must map onto a node with the same degree in U_2 under any isomorphism. One can therefore restrict the potential adjacency matrices to those whose rows and columns are ordered based on the node's degree d. The number of potential adjacency matrices is now $\prod_i |V_i|!$. Of course, this approach is not helpful if all nodes have the same degree (e.g. complete graphs).

Another advanced approach to iteratively classify vertices using basic adjacency information is provided by [17]: First the set of nodes is partitioned into an ordered collection of subsets of V, denoted by $V_1, V_2, ...V_k$ based on an invariant such as the vertex degree. Then for each vertex u, a list $L_u = (a_1, a_2, ...a_k)$ is assigned where $a_i (1 \leq i \leq k)$ is the number of vertices in subset V_i adjacent to u. Using the adjacency information provided by the lists, the original subsets V_i can now be further subpartitioned into ordered collections of subsets

consisting of all vertices with a given list. This refinement can be reapplied until no further refinement can be achieved (*i.e.*, when all subsets contain only nodes with identical lists). Multiple invariants can be used to further refine the vertex classification until the number $\prod_i |V_i|!$ becomes small enough and one can switch to the brute force enumeration of all mappings. Using these techniques makes the determination of graph isomorphism practical in many cases.

In summary, isomorphic subgraphs share a bijective graph code called canonical label. For a graph with unique node labels, a canonical label can be obtained by simply writing down its adjacency matrix. For a graph without node labels however, one can enumerate all potential adjacency matrices and take, *e.g.*, the smallest one as its canonical label. The techniques to classify vertices, such as the iterative vertex classification, can be used to reduce the number of adjacency matrices that have to be considered from $O(|V|!)$ to $\prod_i |V_i|!$.

For the clone detection problem, the existence of node and edge labels comes in as a major advantage. Sometimes however, the labels within a subgraph are not unique and it makes sense to apply the vertex classification techniques.

3.2.2.4. vSiGraM

vSiGraM[16] (Vertical Single Graph Matching) is an exact graph matching algorithm that is very similar to *eScan*. Its goal is to find frequent subgraphs within a single graph by traversing a lattice of frequent patterns in a depth first manner.

At first glance, the descriptions of vSiGraM and *eScan* seem identical. It is only a different use of terminology that makes them different. For clarity, we have reformulated the vSiGraM algorithm description to make terminology consistent with the rest of this work. (For readers interested in the original description[16]: Kuramochi *et al.* use the term “subgraph” instead of canonical label and “embedding” instead of subgraph. An embedding is an instance of an “abstract” subgraph.)

The description of vSiGraM uses the following notation:

- s_k : a subgraph of $\mathcal{G}$ with k edges
- $\mathcal{S}_{k+1}$: a set of subgraphs with k+1 edges
- $s.label$: returns the canonical label for a subgraph s
- cl_{k+1}: a canonical label for a subgraph with k+1 edges
- $\mathcal{CL}_{k+1}$: a set of canonical labels for a set of subgraphs with k+1 edges
- cl^f: a *frequent* canonical label
- $\mathcal{CL}^f$: a set of *frequent* canonical labels
- $\mathcal{G}$: the model graph
- $\mathcal{M}(cl_k)$: the list of subgraphs having the canonical label (cl_k)

```
1  vSiGraM(𝒢, minFreq)
2    𝒞ℒ^f ← ∅
3    𝒞ℒ_1^f ← all frequent canonical labels of size 1 subgraphs in 𝒢
4    for each cl_1^f in 𝒞ℒ_1^f do
5      ℳ(cl_1^f) ← all subgraphs with canonical label cl_1^f
6    end for
7    for each cl_1^f in 𝒞ℒ_1^f do
8      𝒞ℒ^f ← 𝒞ℒ^f ∪ vSiGraM-EXTEND(cl_1^f, 𝒢, minFreq)
9    end for
10 return 𝒞ℒ^f
11
12 vSiGraM-EXTEND(cl_k^f, 𝒢, minFreq)
13   𝒮_{k+1} ← ∅
14   𝒞ℒ_{k+1} ← ∅ // stores new, distinct canonical labels
15   for each s_k in ℳ(cl_k^f) do
16     𝒮_{k+1} ← 𝒮_{k+1} ∪ {s_k ⊕ e | e ∈ E} //extend subgraph with an edge
17   end for
18   for each s_{k+1} in 𝒮_{k+1}
19     𝒞ℒ_{k+1} ← 𝒞ℒ_{k+1} ∪ (s_{k+1}.label) //only distinct canonical labels
20     ℳ(s_{k+1}.label) ← ℳ(s_{k+1}.label) ∪ {s_{k+1}} //add subgraph
21   end for
22   for each cl_{k+1} ∈ 𝒞ℒ_{k+1} do
23     if cl_k^f is not the generating parent of cl_{k+1} then
24       continue
25     end if
26     compute cl_{k+1}.freq from ℳ(cl_{k+1})
27     if cl_{k+1}.freq < minFreq then
28       continue
29     end if
30     𝒞ℒ^f ← 𝒞ℒ^f ∪ {cl_{k+1}} ∪ vSiGraM-EXTEND(cl_{k+1}, 𝒢, minFreq)
31   end for
32 return 𝒞ℒ^f
```

Listing 3.2: vSiGraM algorithm

A canonical label is considered *frequent* (superscript f) if the number of subgraphs sharing the canonical label is above a minimum frequency threshold ($minFreq$).

As vSiGraM is a graph matching algorithm, the goal is to find subgraphs occurring in the model graph with a minimum required frequency. In the context of clone detection, a subgraph occurring twice would be considered frequent. vSiGraM starts by enumerating all canonical labels of frequent size 1 subgraphs occurring in the model (line 3) and stores for each canonical label the associated subgraphs (lines 4-6). In lines 7-9, each canonical label for size 1 subgraphs is input into the vSiGraM-EXTEND function, starting the depth first traversal of the pattern lattice.

eScan, in contrast to vSiGraM, does not perform a DFS for each frequent canonical label but for each frequent subgraph.

vSiGraM-EXTEND uses the subgraphs of size k associated with each frequent canonical label ($\mathcal{M}(cl_k^f)$) to build subgraphs of size k+1 (lines 15-17). The extension operation $\oplus$ works just like in *eScan*. In line 18, $\mathcal{S}_{k+1}$ contains all subgraphs that can be obtained by the addition of one edge to a subgraph having the canonical label cl_k^f. $\mathcal{CL}_{k+1}$ stores all distinct canonical labels associated with all subgraphs in $\mathcal{S}_{k+1}$ in line 19. In line 20, subgraphs are added to the list of subgraphs having the canonical label $s_{k+1}.label$.

To avoid a repeated processing of a canonical label, vSiGraM uses a generating parent strategy. The DFS for a canonical label cl_{k+1} continues only if cl_k^f is the generating parent of cl_{k+1}. The strategy for determining the generating parent relationship between two canonical labels is slightly different from the approach used in *eScan*. Again, the canonical label cl_{k+1} defines a unique order on the edges of its subgraphs. If the last edge in this order that does not disconnect the corresponding subgraphs is removed, we obtain another canonical label L_k. If L_k is identical to cl_k^f, cl_k^f is the generating parent of cl_{k+1}.

If the number of subgraphs with canonical label cl_{k+1} is above $minFreq$, cl_{k+1} is considered frequent. cl_{k+1} is added to the set of frequent canonical labels $\mathcal{CL}^f$ and used for a the next recursive call to vSiGraM-EXTEND. The DFS backtracks if vSiGraM cannot find 'bigger' frequent canonical labels.

As in *eScan*, vSiGraM finds frequent subgraphs that overlap, *i.e.*, that have vertices in common. To overcome this problem, Kuramochi *et al.* build an overlap graph in which each subgraph is represented as a node and there is an edge between the nodes if the corresponding subgraphs have a node in common. As a next step, the maximum independent set (MIS) of the overlap graph is determined. The solution to the MIS problem yields the biggest possible set of non-overlapping subgraphs. The MIS problem can be solved directly by exact or approximate algorithms. Finding the MIS on a graph G is equivalent to finding the maximum clique on G's complement graph $\overline{G}$ which contains the same node set as G and an edge between two nodes if the corresponding subgraphs do not overlap. This is the approach taken by *eScan*. However, vSiGraM calculates a subgraph's frequency (without overlaps) in each iteration. *eScan*, in contrast, checks for each candidate fragment if it overlaps with the current fragment and performs the overlap analysis only once at the end.

Kuramochi *et al.* also present an overview of further algorithms for frequent pattern identification in single graphs: hSiGraM traverses the pattern lattice using a breadth first strategy trying to find bigger isomorphic subgraphs by joining smaller isomorphic subgraphs. *Grew* is a heuristic approach that collapses frequent patterns into a single *multivertex* during search.

3.2.3. aScan

aScan is a clone detection algorithm that identifies exact and approximate clones within a model graph. Two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ of G that are not necessarily isomorphic are still considered similar if they in turn contain two "relatively big" isomorphic subgraphs $G'_1 = (V'_1, E'_1) \subseteq G_1$ and $G'_2 = (V'_2, E'_2) \subseteq G_2$. Note that G_1 and G_2 are also considered similar if they are isomorphic. Thus, approximate similarity subsumes exact similarity.

aScan uses a vector-based approximation of the structure with a subgraph called Exas[4]. A subgraph is represented by a characteristic vector containing the occurrence counts of specific structural features. Using this approach, the distance between two subgraphs can be measured by a vector distance function (*e.g.*, the 1-norm). Two subgraphs are considered similar if the distance between their characteristic vectors is small enough.

A characteristic vector consists of the occurrence counts of two kinds of structural features: *(p,q)-nodes* and *n-paths*. A (p,q)-node is a node having p incoming and q outgoing edges. An n-path is a sequence of n nodes that are connected by a directed path in the subgraph. A concrete (p,q)-node is depicted by the node label and its in and out degree, *e.g.*, "Gain-1-1". An n-path is depicted by the sequence of node and edge labels along the directed path, *e.g.*, "Gain-Sum-Mul".

To efficiently describe a subgraphs feature set, *aScan* maintains a global list of all available features used by all vectors. A subgraph's characteristic vector contains the occurrence count of a specific feature at the index position a feature has in the global list. As the characteristic vector of a subgraph contains many 0-entries (*i.e.*, the feature does not occur in the subgraph) it makes sense to use a sparse vector implementation.

```
1   function aScan(G = (V,E,T))
2     k ← 1, L_1 ← E
3     repeat
4       k ← k + 1
5       C_k ← C_k ∪ {f_{k−1} ⊕ e | f_{k−1} ∈ L_{k−1}, e ∈ L_1}
6       for i = l(k) to k − 1 do C_k ← C_k ∪ L_i
7       CG ← CG ∪ Group(C_k)
8       Filter(CG)
9       L_k ← { all detected cloned k−fragments }
10    until L_k = ∅
11  return CG
```

Listing 3.3: Approximate clone detection

An *aScan* implementation traverses the clone lattice in a breadth-first manner. It starts by collecting all edges existing in G into L_1. In each of the following iterations, cloned fragments from layer L_{k-1} are used to build fragments of size k. Just like in *eScan* and vSiGraM, the *extension operation* $\oplus$ is used to create bigger fragments by adding an edge. A fragment's characteristic vector can be computed in an incremental manner using the smaller fragment's characteristic vector as basis[4]. In line 5, all extended fragments of size k are added to the current set of potentially cloned fragments C_k. As for approximate

clones a fragment of size k may be similar to a fragment with a different size, clones of smaller sizes are added to C_k in line 6. $l(k)$ returns the minimum size a fragment of size x needs to have in order to be potentially similar to a fragment of size k ($x < k$). After all candidate fragments have been collected into C_k, *aScan* groups cloned fragments by comparing their characteristic vector distances (line 7). To avoid all pairwise comparisons, a fragment's characteristic vector is input into a locality sensitive hashing function [18] in order to distribute similar fragments into the same hash bucket. As a result, only the fragments within a bucket have to be compared pairwise. Two fragments are in the same clone group if their vector distance is below a certain threshold. A filtering step partitions overlapping fragments within a clone group by building an overlap graph and applying a maximum clique cover algorithm. This step is identical to the procedure used by *eScan*. All identified cloned fragments are collected into L_k and used for the next iteration. When L_k is empty, the detection is finished and the set of all identified clones CG is returned.

As an optimization, *aScan* prevents the extension operation using an edge e after the extension of a fragment using e did not result in a cloned fragment.

3.2.4. ModelCD Optimizations

As the number of generated candidate subgraphs can in theory become very high, Pham *et al.* propose two performance optimizations that address subsystems and high degree nodes.

When ConQAT creates the graph representation of a Simulink model it *flattens* subsystems, *i.e.*, all blocks wired to corresponding in- and out-ports are connected directly and the subsystem block is removed. In other words, flattening a subsystem is the inverse operation of creating a subsystem and the resulting model graph is flat. Pham *et al.* propose an alternative strategy for dealing with subsystems: Subsystem blocks shall be kept as a node within the model graph and the contents of the subsystem shall be added to the model graph as a disconnected component. The authors suggest that subsystems with the same Simulink block name should be reported as cloned subsystems (Nguyen Anh Hoan, personal communication, August 16, 2009). For a group of cloned subsystems, only the contents of one clone instance are added to the model graph as a disconnected component while the substructure of all other subsystem clone instances are ignored.

As nodes with a high degree generate a high number of subgraphs during the extension operation, they are separated from the original model graph G at the beginning. The result of this step is a new model graph G' that may contain more disconnected components than G. The algorithm is then executed on G' and yields a set of clone groups CG. As a next step, a new model graph N is built that consists only of the high degree nodes that were separated at the beginning and a node for each clone contained in the set of clone groups CG. All clone-nodes of a clone group carry the same node normalization label. There is an edge between a high degree node and a clone node if there was an edge between one of the nodes belonging to a clone and a high degree node in the original model graph G. There may also exist edges connecting two high degree nodes. The discovery is then executed using the *eScan*/*aScan* algorithm on N in order to find bigger clones.

3.3. ConQAT's Model Clone Detection

To our knowledge, this algorithm was the first published algorithm for clone detection in graph-based models. As Deissenboeck *et al.* did not name their algorithm in [3], we will refer to this algorithm as ConQAT's Model Clone Detection (CMCD) for the remaining parts of this work.

During preprocessing of the model graph, all subsystems are flattened, *i.e.*, all blocks connected to an inport or outport of a subsystem are joined directly to the corresponding block within the subsystem. The resulting graph thus has no longer a hierarchic structure.

CMCD runs in two conceptionally distinct phases: First, all clone pairs are identified. In a second step, clone pairs are clustered to form clone groups with potentially more than two clone instances.

The pair detection of CMCD is outlined in Listing 3.4. It iterates over all possible pairings of nodes and uses pairs with the same label as the starting point of a breadth-first search (BFS) to identify maximal clones. D denotes a set of already visited node pairs, S a set of nodes seen in the current BFS, and C contains all node pairs of the current clone.

```
1  D := ∅
2  for each (u,v) ∈ V × V with u ≠ v ∧ L(u) = L(v) do
3    if {u,v} ∉ D then
4      Queue Q := {(u,v)}, C := {(u,v)}, S := {u,v}
5      while Q ≠ ∅ do
6        dequeue pair (w,z) from Q
7        from the neighborhood of (w,z) build a list of
8        node pairs P with equivalent labels
9        for each (x,y) ∈ P do
10         if (x,y) ∈ D then continue with loop at line 2
11         if x ≠ y ∧ {x,y} ∩ S = ∅ then
12           C := C ∪ {(x,y)}, S := S ∪ {x,y}
13           enqueue (x,y) in Q
14     report node pairs in C as clone pair
15     D := D ∪ C
```

Listing 3.4: CMCD clone pair detection

In line 7, from the neighborhood of the pair (w,z) a single mapping of nodes is used to construct a list P of node pairs. In order for two nodes to be mapped, they must have the same node label and be connected to the rest of the clone via two edges with the same direction and edge label. CMCD considers only one potential mapping of nodes as pursuing other node mappings would have a negative impact on the algorithm's complexity. In order to choose "the right" node mapping, CMCD uses a similarity function $\sigma : V \times V \rightarrow [0,1]$ that yields a measure for the similarity of two nodes. Fur this purpose, σ also considers the proximate neighborhood of two nodes. Identifying the best node mapping corresponds to a weighted maximal bipartite graph matching using the similarity values as weights. Line 9 is optional and continues the iteration with the next node pair if a node pair has already been discovered in a previous iteration. This is supposed to prevent similar clones with

different extensions. If two nodes (x,y) in a node pair are distinct and have not already been detected in the current iteration, they are added to the currently processed clone. (x,y) are added into Q so that their neighborhood will be explored in a following iteration. When Q is empty, all nodes in C are reported as a clone pair and added to the set of already discovered node pairs D.

This step yields only clone pairs. In order to build clone groups with more than two members, all clone pairs are clustered based on only their node set. In contrast to ModelCD, CMCD thus does not require all members of a clone group to be isomorphic. ConQAT uses a union-find structure to build clone groups on-line. While clones in a clone pair can not possibly overlap, it is possible that clones in a clone group overlap.

A more detailed description of CMCD can be found in [3].

3.4. gSpan

gSpan (graph-based Substructure pattern mining) is a graph matching algorithm that assigns a unique DFS code for each subgraph which serves as its canonical label. The DFS codes can be used to define a lexicographic order among graphs and to build a DFS code tree. Through a depth first search within the DFS code tree all frequent subgraphs can be discovered. More information on *gSpan* can be found in [19].

4. Improving clone relevance

In Chapter 2, we outlined that the similarity between model clones can be defined in multiple ways. The algorithms we presented in Chapter 3 use subgraph isomorphism to measure the similarity between subgraphs. This approach, however, only approximates semantic similarity, *i.e.*, the property that two or more subgraphs express the same domain concept. Unfortunately, subgraph isomorphism is not suited to identify all semantic clones as there often exist multiple ways to describe the same idea. There are also cases in which isomorphic subgraphs describe a different idea, *e.g.*, when vital information was removed during normalization.

Furthermore, not all clones existing within a model are also relevant to a person performing a clone detection. The relevance of a clone depends on a variety of circumstances. We will mention several of these factors in the following section. The available algorithms for clone detection identify many clones that, although they fulfill the mathematical requirements, are not interesting to developers.

The search for relevant clones can be quantified in terms of the performance measures *recall* and *precision* used in information retrieval [20]. The recall is maximized by the identification of all existing relevant clones and the precision is maximized by reporting only relevant clones. In this chapter, we describe approaches that increase the precision by filtering those clones that fulfill the isomorphism relation but are not interesting for developers. Figure 4.1 gives a graphical overview of the clone set relations. Our goal is to minimize the yellow part that stands for the identified, yet uninteresting clones.

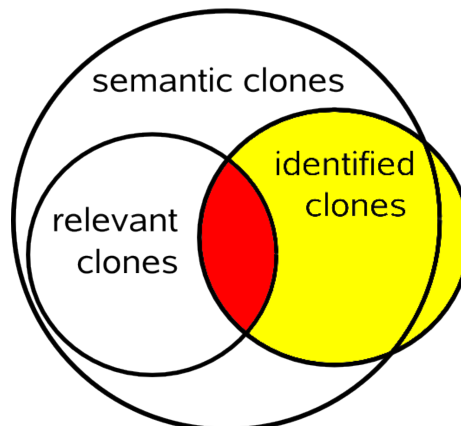


Figure 4.1.: Clone set relations

Furthermore, we study approaches how clones can be ranked *w.r.t.* their relevance.

4.1. Factors affecting clone relevance

Hørland & Sejer Christensen define that “something is *relevant* to a task if it increases the likelihood of accomplishing the goal which is implied by the task”. Therefore, to identify the clones that are relevant to a person performing the clone detection, one needs to know that person’s aims. A number of use cases for graph-based model clone detection have been identified that led to the initial development of a model clone detection algorithm:

1. Identification of libraries:
   Which duplicate elements can be externalized?

2. Impact Analysis:
   Should a change also be conducted in other parts of the model?

3. Metric for model maintainability:
   How much effort is required to maintain and evolve a model?

4. Identification of product line elements:
   Which common elements are used across products?

5. Rule checking:
   Does a model contain forbidden structures?

6. Plagiarism:
   Does a model infringe intellectual property rights?

There are cases in which a clone is relevant only *w.r.t.* a specific use case: A developer would, *e.g.*, like to be informed about a cloning relationship when he is making changes to one instance, while at the same time he would not externalize the clone into a library. While it is important to keep these different goals in mind, the scope of this work is limited to measures that can be applied for all use cases and focuses primarily on the identification of library elements.

To make things worse, the relevance of a clone may be assessed differently amongst developers. Furthermore, the relevance of a clone may depend on the model under analysis, *e.g.*, a small clone may be relevant if no bigger clones exist.

4.2. Assessing clone relevance in CMCD

CMCD already supports basic means to assess a clone’s relevance: For each node in the model graph, a user-defined *node weight* value is assigned that depends on the node’s block type. Node weights can be used as a basic tool to distinguish between clones that contain “interesting” block types and clones that contain uninteresting block types.

To sort clones in a clone report CMCD considers the following clone group metrics:

- Clone node size:
  The number of nodes within each clone instance of a group

- Clone weight:
  The sum of node weights within each clone instance of a group

- Clone group weight:
  The sum of node weights of all clone instances within a group
- Clone group cardinality:
  The number of clone instances within a group

Clones that do not fulfill a minimum required node size or weight are not reported.

While these measures provide a basis for the assessment of clone relevance, the output still contains clones that are considered irrelevant *w.r.t.* to the identification of new library elements: Clones that would not be externalized because they do not correspond to a domain concept. Furthermore, the ranking of clones based on the metrics listed above gives only little indication of how difficult or promising the externalization of a clone could be.

4.3. Graph distance filters

It has been suspected that graph theoretical properties could be helpful in deciding whether a clone is relevant [3]. In order to further analyze this assumption, we examined a set of clones from our industry partner. These clones were classified as relevant or irrelevant. Using this training set, we studied the clones' graphs *w.r.t.* a number of graph theoretic metrics: We evaluated a number of graph distance properties, detailed in [21], such as the diameter of a clone as well as general properties such as the number of nodes and edges. Using these metrics, we developed three filters that identified a high percentage of uninteresting clones within the training set. All filters work on a clone group's first clone instance and do not take into account the direction of edges.

4.3.1. Diameter filter

This filter removes clones having an "extreme" diameter *w.r.t.* their node count, *i.e.*, for a clone graph C to *pass* this filter, the following condition must hold:

$$\sigma < \frac{diameter(C)}{nodeCount(C)} < \delta$$

This filter aims at removing clones whose structure is very similar to a single path (for which a clone graph's diameter equals the node count minus one) or similar to a fully connected graph. For our experiments, we used the thresholds $\sigma = 0.1$ and $\delta = 0.9$.

4.3.2. Average distance filter

This filter was designed to remove clone graphs in which the shortest paths in-between nodes are relatively long, *e.g.*, for graphs consisting of only one path. It is thus also a measure for the connectedness of a graph. As mentioned in Chapter 2, the eccentricity ($ecc(v)$) of a node is the length of the longest shortest path between v and all other nodes in the graph. For a clone graph to *pass* this filter, the following condition must hold:

$$\frac{\sum\limits_{v \in V} ecc(v)}{nodeCount(C)^2} > \sigma$$

In other words, this filter calculates the average eccentricity in the clone graphs divided by the number of nodes in the clone graph. For our experiments we used a σ value of 0.5.

4.3.3. Zero-weight filter

This filter does not use a graph distance measure but removes clones that have a high percentage of nodes with a weight of 0. For a clone to *pass* this filter, the following condition must hold:

$$\frac{|\{v \in V | v.\text{weight} = 0\}|}{|V|} < \sigma$$

For our experiments, we used the threshold $\sigma = 0.5$

4.4. Clone ranking scheme

While the use of filters aims at the removal of irrelevant clones, our industry partner had a slightly different view on the approach: Our partner was particularly interested in being notified about highly relevant clones. For this purpose, we developed a new clone group metric that could be used to rank clone groups within the output. We deemed the "clone group weight" (W_{CG}), *i.e.*, the cumulated weights of all clone instances within a clone group, the most important metric to assess the relevance of a clone. However, we wanted to further qualify this metric by using additional information. We therefore created the *modified weight of a clone group* (MW_{CG}). The MW_{CG} value is calculated by multiplying a group's cumulated weight with a ranking value (RV) out of $[0, 1]$.

$$MW_{CG} = W_{CG} \cdot RV$$

We will detail two potential ranking values (RVs) in the following sections. Multiple ranking values could be combined, *e.g.*, by using their (weighted) mean.

4.4.1. Interface ranking value

The complexity of removing a clone from a model by externalizing duplicate computations into a library is possibly influenced by the number of edges that connect nodes within the clone instance to nodes outside the clone instance. We will refer to these edges as the *interface edges* and the number of interface edges as the *interface size* of a clone instance. The interface size is not necessarily equal for all clone instances within a clone group.

When a clone instance is replaced by a reference to a library element, the interface size defines the number of Simulink lines that have to be newly established for each clone instance. Potentially, clones having a smaller interface size are better candidates to form a library element, as their externalization is less likely to introduce faults. If a clone with a relatively small interface is used to create a library element, this would comply with the design principle of "low coupling" between subsystems.

The interface ranking value for a clone group CG (RV^i_{CG}) is computed as follows: Let $c \in CG$ denote a clone instance, $c.interfaceSize$ the interface size of clone instance c and $c.nodeCount$ the number of nodes within c.

$$RV^i_{CG} = \frac{\sum_{c \in CG} MAX(1 - \frac{c.interfaceSize}{c.nodeCount}, 0)}{|CG|} \in [0, 1]$$

4.4.2. Relative weight ranking value

The effort required to externalize a clone is possibly influenced by the number of nodes within the clone, *i.e.*, it may be easier for a developer to catch the semantics of a clone if it consists of fewer blocks. Consequently, if two clones have the same cumulated weight value but a different number of nodes, it may be that the externalization of the clone with the fewer nodes has a better cost-benefit ratio than the externalization of the clone with more nodes.

To quantify this circumstance, we calculate for each clone group a ranking value RV_{CG}^{w} that describes its ratio of weight to nodes relative to the clone group with the highest ratio of weight to nodes identified in the detection:

$$RV_{CG}^{w} = \frac{\frac{W_{CG}}{CG.nodeCount}}{MAX(\forall_{Group \in CloneGroups} : \frac{W_{Group}}{Group.nodeCount})}$$

4.5. Ranking scheme evaluation

In order to evaluate the proposed ranking scheme, we performed a case study with our industry partner. We chose a set of 21 clone groups, each consisting of two clone instances and asked two developers to rank the clone groups *w.r.t.* their appropriateness to form a library element. We did not require a total order among the groups, *s.t.* the developers could assign the same position within the ranking to two or more clone groups. The clones' average node count is approx. 19 with a minimum of 7 nodes and a maximum of 105 nodes. More information about the clone groups used for the case study can be found in Appendix A.

To analyze the appropriateness of a variety of clone metrics, we compared the developer's ranking to the ranking based on the clone metrics. For each developer and metric, we calculated for each clone group the difference between the developer's rank and the rank based on the metric and summed up the absolute value. Let R_{CG}^{d} denote the rank that developer d assigned for clone group CG and R_{CG}^{m} the rank a clone group CG has in the ranking that is based on the metric m. We then calculated for each developer and metric the deviation $D_{d,m}$ as

$$D_{d,m} = \sum_{CG \in CloneGroups} |R_{CG}^{d} - R_{CG}^{m}|$$

We used a fractional ranking for equal values, *i.e.*, the rank of clone groups that compare equal is the mean of their ordinal rankings. For example, if two items were equal and would receive the ranks 1 and 2 under an ordinal ranking, we assigned the rank 1.5 to both items and rank 3 to the following item.

We used two Simulink block weight assignment schemes: ConQAT's default Simulink block weights and a weight assignment of our automotive industry partner. However, as the results of both weighting schemes are comparable and for the sake of clarity, we will only discuss the results of the automotive node weight assignment here. The results of the default weighting scheme can be found in Appendix A.

We computed the deviations $D_{d,m}$ for both developers and for the following clone group metrics:

- Clone node size
- Clone group weight
- Modified clone group weight (Interface ranking value)
- Modified clone group weight (Relative weight ranking value)
- Interface ranking value
- Relative weight value

The results of our study can be found in Section 6.2.

5. Evaluation of model clone detection algorithms

This chapter contains the findings of our model clone detection algorithm evaluation. We assess both ModelCD algorithms, ConQAT's Model Clone Detection, and *gSpan w.r.t.* to their appropriateness for large-scale models. This chapter is structured as follows:

Section 5.1 introduces the Simulink models we used for our assessment.

Section 5.2 details our findings for ModelCD and mentions potential improvements for identified problems. This section contains four subsections: We discuss the impact of the ModelCD optimization strategies in Section 5.2.1. Section 5.2.2 details weaknesses of *eScan* and proposes potential improvements. Section 5.2.3 contains our judgment on the appropriateness of *aScan* for large-scale systems. Finally, in Section 5.2.4, we discuss improvements to the clone grouping approach used by ModelCD.

In Section 5.3, we describe a new approach for the detection of cloned subsystems based on subsystem contents.

Section 5.4 mentions weaknesses of ConQAT's Model Clone Detection. Section 5.5 contains the experiences we gained when using the graph mining algorithm *gSpan* for model clone detection.

Finally, Section 5.7 summarizes the findings of this chapter.

5.1. Models used for algorithm evaluation

In order to compare our results with the ModelCD case study in [5], we used the same four models for our analysis. These models are publicly accessible on MATLAB Central: A Simulink model for a communications lab (SIM), a simulation of multiple unmanned air vehicles (MUL), a video surveillance system (SEM) and an echo canceler model (ECW) used to introduce beginners to modeling in Simulink. Furthermore, we also used a gas separation plant model (MPC) and a comparatively very large model from our automotive engineering industry partner (AUT).

Model	# Nodes	# Edges
SIM	428	415
MUL	475	576
SEM	1741	2029
ECW	2312	2274
MPC	369	395
AUT	98251	90056

Table 5.1.: Overview of evaluated Simulink models

5.2. ModelCD

For our assessment of graph-based model clone detection algorithms, we integrated the ModelCD algorithms *eScan* and *aScan* into ConQAT. As the authors did not make their source code available upon request, we implemented the algorithms based on the description in the corresponding publications [5, 4]. This section contains the experiences we gained during the application of ModelCD to the models introduced in the previous section.

5.2.1. ModelCD Optimizations

In order to reduce time and memory requirements of *eScan* and *aScan*, Pham *et al.* propose two optimization techniques: A special treatment of nodes with a high degree and a detection of cloned subsystems. In this subsection we will detail the advantages and downsides of these strategies.

5.2.1.1. Separation of high degree nodes

Before clone detection starts, nodes with a high degree (*switches*) are separated from the model graph G. This step results in a new graph G'. After a clone detection algorithm (*eScan* or *aScan*) has been carried out on G', the identified clone groups, together with the switches are used to construct a new graph N. As a next step, a clone detection is performed on N, which yields bigger clone groups that are composed of smaller clone groups and switches.

Impact on detection completeness

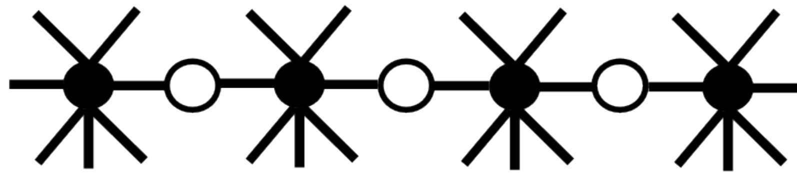
Using this technique prevents the detection of clones that are composed of an alternating sequence of high and low degree nodes if the low degree nodes are not part of a clone instance. Figure 5.1(a) shows an example of a clone instance that can not possibly be detected due to the special treatment of high degree nodes.

A cloned fragment may be reported as multiple smaller fragments if these fragments are connected to each other over a path that consists of a single node surrounded by only two high degree nodes such as the model in Figure 5.1(b). Partial fragments may not even be reported if they fall below the minimum required size (or weight).

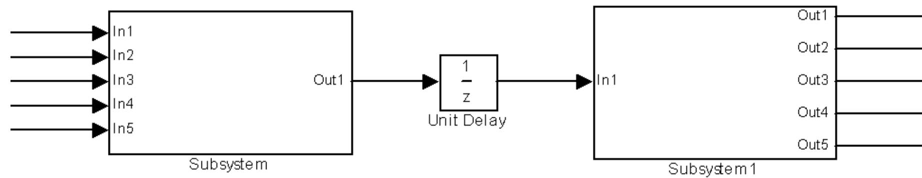
Although structures as presented in Figure 5.1 may seem unlikely at first glance, it is important to know that for the ModelCD case study, all nodes with a degree above 6 were treated as high degree nodes (Nguyen Anh Hoan, personal communication, August 16, 2009). When applied to the models of our industry partner, $\frac{1}{5}$ of all nodes and $\frac{1}{3}$ of all edges are separated from the model graph at the beginning. Therefore, the negative impact that this optimization has on detection completeness must not be underestimated.

Complex detection on the clone graph

After the detection has been carried out on the graph without high degree nodes (G'), a new graph N is created. We term this graph N the *clone graph.* The node set of N consists of a node for each clone identified in the clone detection on G' (*clone nodes*) and all high degree nodes (*switches*) that were separated at the beginning. N contains an edge between a switch and a clone node if the original graph G contained an edge between the switch



(a) Missed clone due to high degree node separation



(b) Subgraph that separates a cloned fragment into two smaller fragments

Figure 5.1.: Clone instances not correctly detected due to high degree node optimization

and one of the nodes within the subgraph represented by the clone node. Furthermore, N contains edges between switches if there was an edge between those switches in G.

Unfortunately, in our experiments, the resulting graph N was often dense and thus intractable for both ModelCD algorithms. Switches with a high degree in the original graph had very high degrees in N. The degree is higher, as for each node n in G that is connected to a switch s, potentially each cloned subgraph of G that contains n is connected to s in N.

Regrettably, the high number of clones identified by ModelCD adversely impacts this detection phase. While the removal of covered clone groups subsequent to the clone detection on G' often reduces the number of clone nodes, the partitioning of clone groups into non-overlapping clone groups increases the number of clone nodes in N.
Although a stepwise integration of high degree nodes based on their node labels can decrease the degree of switches in N, *eScan* and *aScan* could not process the clone graph for the models MPC and (parts of) AUT.

5.2.1.2. Removing duplicate subsystems

Pham *et al.* propose that the hierarchic structure of a MATLAB/Simulink model graph should not be converted into a flattened representation. Furthermore, subsystems with the same name should be reported as cloned subsystems. For cloned subsystem groups, only the contents of one clone instance shall be added to the model graph.

The developers of our industry partner agreed that, if an entire subsystem was cloned, they would prefer to be informed about a cloned subsystem rather than its atomic blocks. If only the atomic blocks of cloned subsystems were reported, developers would first have to check manually if the entire subsystem is cloned or not. As mentioned in the introduction, MATLAB/Simulink developers often create subsystems that represent a domain concept. The interface of a well-designed subsystem is usually thin and often documented.

Therefore, externalizing a cloned subsystem into a library element seems to be often easier and usually more meaningful than outsourcing an arbitrarily chosen subgraph of the model.

However, we strongly doubt that a subsystem's block name as the only criterion is suited to decide whether two subsystems are clones of each other. While the approach by Pham *et al.* may have been appropriate for the models used in their case study, it makes little sense for the models of our industry partner: These models contain subsystems with the same name that are in no way clones of each other as well as subsystems with isomorphic contents but a different name. We therefore created a modified approach to identify cloned subsystems, that looks at the contents of subsystems. We will detail this approach in Section 5.3.

If only the contents of one instance of cloned subsystem groups is included in the model graph, a significant reduction of the model size can be achieved for some models (*cf.* Table 5.2). The comparison of Table 5.1(b) with Table 5.1(c) supports the assessment of our industry developers that a subsystem's name is not a good criterion for cloned subsystem identification: Obviously, for SEM and ECW unequal subsystems are considered clones and for MPC no cloned subsystems could be identified as the model contains only identical subsystems with a different name. As a consequence, for multiple executions of the algorithm, the name-based approach may return different results depending on which clone instance is inserted into the model graph.

(a) Prior to removal

Model	# Nodes	# Edges
SIM	428	415
MUL	475	576
SEM	1741	2029
ECW	2312	2274
MPC	369	395
AUT	98251	90056

(b) After content-based removal

Model	# Nodes	# Edges
SIM	387	379
MUL	450	432
SEM	710	704
ECW	1315	984
MPC	294	317
AUT	66944	66026

(c) After name-based removal

Model	# Nodes	# Edges
SIM	398	382
MUL	457	452
SEM	569	542
ECW	1050	791
MPC	369	395
AUT	54321	53286

Table 5.2.: Simulink model sizes before (a) and after the removal of duplicate subsystems based on subsystem contents (b) and subsystem name (c)

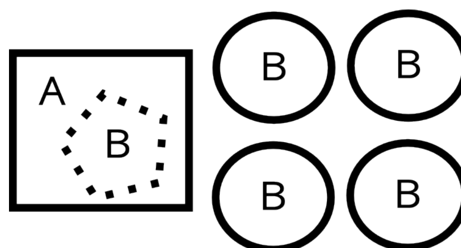


Figure 5.2.: Subgraph B is reported as two clone groups

There is a drawback associated with approach of including only the contents of one cloned subsystem instance into the model graph, depicted in Figure 5.2: Subsystem 'A' contains a subgraph 'B'. Subgraph 'B' also occurs in 4 other subsystems. However, subgraph B within subsystem A is not a subsystem. As the contents of only one instance of subsystem B are added to the model graph, two different clone groups are identified:

1. A group of cloned subsystems containing B with 4 members
2. A group containing two cloned subgraphs of type B in which one instance corresponds to the subgraph B within subsystem A and the other instance corresponds to the contents of a single clone instance of the cloned subsystem group that were included into the model graph

If, *e.g.*, a person performing the clone detection only looks at the second clone group, he is not aware that subgraph B actually occurs 5 times within the model.

5.2.2. eScan

If our *eScan* implementation is executed without the optimizations described in Section 3.2.4, it does not terminate within reasonable time and space boundaries for all models listed in Table 5.1, except SIM. Even with both ModelCD optimizations, *eScan* can not process the models MPC and AUT.

During *eScan*'s clone detection each cloned subgraph of the model and all of their edge extensions have to processed by *eScan*. At the worst, all subgraphs of the model graph have to be created, *i.e.*, in its worst case the algorithm has exponential complexity.

What exactly makes a graph (in)tractable for *eScan*? The overall runtime of the *eScan* discovery function is determined by the number of fragments created during its DFS. As the amount of created fragments depends primarily on the number of isomorphic subgraphs within a model graph, we are unaware of a means to judge upon the complexity of an input graph beforehand, *i.e.*, without actually performing a clone detection on it.

We conclude that the number of nodes and edges of a graph is of limited significance concerning the scalability of *eScan*.

5.2.2.1. Reducing the number of fragments created

As the runtime of *eScan* is primarily affected by the number of processed fragments, we tried to reduce the number of fragments considered during clone detection. We noticed that the original *eScan* function *Discover*, depicted in Listing 5.1, creates a large number

of unnecessary fragments. In lines 2-3 of Listing 5.1, all potential fragments of size $k+1$ are created that can be obtained by extending a fragment in $Clones(f_k)$. However, *eScan*'s generating parent function will return *false* for all fragments that were not created through an extension of f_k.

We therefore created an adapted version of the *Discover* function (Listing 5.2) that first creates all extensions of f_k and checks the *eScan* generating parent condition (line 4). Only if f_k is the generating parent of f_{k+1}, the fragments within $Clones(f_k)$ are extended. However, this extension operation is restricted to edges that have the same edge label as the edge used for the extension of f_k that yielded f_{k+1}.

```
1  function Discover(f_k, Clones(f_k))
2    for each g_k ∈ Clones(f_k) do
3      C_{k+1} ← C_{k+1} ∪ {g_k ⊕ e | e ∈ E}
4    for each c_{k+1} ∈ C_{k+1} do
5      if GeneratingParent(c_{k+1}) = f_k then
6        Find(Clones(c_{k+1}))
7        if |Clones(c_{k+1})| > 1 then
8          L_{k+1} ← L_{k+1} ∪ Clones(c_{k+1})
9          Discover(c_{k+1}, Clones(c_{k+1}))
```

Listing 5.1: Original *eScan* function Discover

```
1  function Discover(f_k, Clones(f_k))
2    F_{k+1} ← F_{k+1} ∪ {f_k ⊕ e | e ∈ E}
3    for each f_{k+1} ∈ F_{k+1} do
4      if GeneratingParent(f_{k+1}) = f_k then
5          for each g_k ∈ Clones(f_k) do
6            C_{k+1} ← C_{k+1} ∪ {g_k ⊕ e | e.label = f_{k+1}.lastEdge.label}
7        Find(Clones(f_{k+1}))
8        if |Clones(f_{k+1})| > 1 then
9          L_{k+1} ← L_{k+1} ∪ Clones(f_{k+1})
10         Discover(f_{k+1}, Clones(f_{k+1}))
```

Listing 5.2: Adapted *eScan* function Discover

By extending fragments in $Clones(f_k)$ only with edges having a specific label, the creation of candidates with other edge labels is avoided. If f_k is not the generating parent of f_{k+1}, no other fragment in $Clones(f_k)$ has to be extended. Of course, not creating a fragment also circumvents the expensive computation of a canonical label for it.

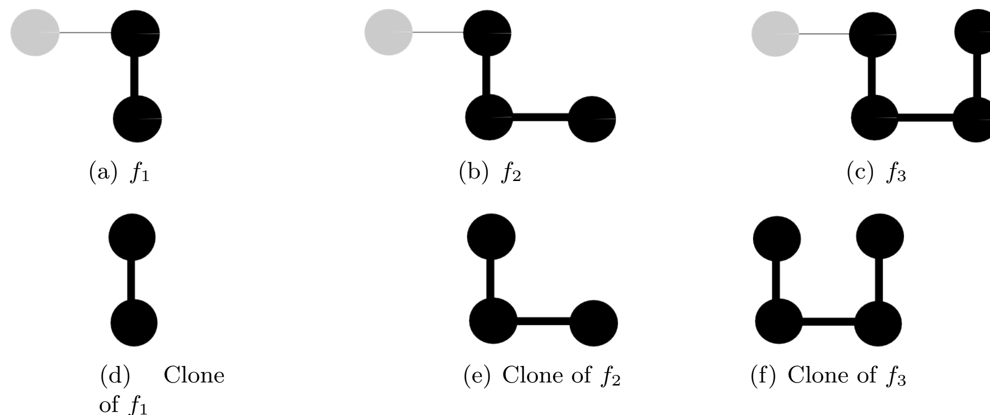
In our adapted algorithm depicted in Listing 5.2, a fragment is generated multiple times if F_{k+1} contains multiple fragments that were created by the extension with edges with the same label. This can easily be avoided by using a shared candidate set for each distinct edge label. However, to avoid confusion, we did not include this detail in the description of the modified algorithm.

In vSiGraM's extend operation, all potential candidate subgraphs are created before checking the generating parent condition. However, as the generating parent strategy is different, no unnecessary subgraphs are created by vSiGraM.

An important disadvantage of *eScan* compared to vSiGraM is that *eScan* discovers a clone group multiple times during the DFS, once for each clone instance within the group. The design of *eScan* however removes the need for a frequency calculation within each iteration, as it is done in vSiGraM. To use vSiGraM for clone detection however, an easier frequency calculation would suffice which only checks that at least two non-overlapping subgraphs exist for a canonical label.

5.2.2.2. Avoiding the repeated extension of unsuccessful edges

If for a cloned fragment f_k, the extension with an edge e does not yield a cloned fragment of size $k+1$, all other extensions within the offspring set of f_k will not be able to create a cloned fragment by including the (*unsuccessful*) edge e. However, in the original *eScan* algorithm, unsuccessful edges are used again and again for expansion by all fragments in the offspring set. See Figure 5.3 for an example in which e is depicted as the edge leading to a grey node. There is no need for f_2 and f_3 to perform the extension operation using e after it has already proven unsuccessful for f_1.



(a) f_1 (b) f_2 (c) f_3

(d) Clone of f_1 (e) Clone of f_2 (f) Clone of f_3

Figure 5.3.: An unsuccessful edge is used multiple times for expansion of a fragment

For our implementation of *eScan*, when the extension of a fragment f_k with an edge e does not yield a cloned fragment, we prohibited the usage of e for the extension operation of all offspring fragments of f_k. However, all edges that were unsuccessful during the expansion of f_k have to be identified before the depth first search continues. Listing 5.3 shows our adapted discovery function that avoids the repeated extension of unsuccessful edges. Each fragment f_k on the current DFS path has associated with it a set of unsuccessful edges, $f_k.Pro$.

```
1   function Discover(f_k, Clones(f_k))
2     F_{k+1} ← F_{k+1} ∪ {f_k ⊕ e | e ∈ E \ f_k.Pro}
3     Pro ← f_k.Pro
4     for each f_{k+1} ∈ F_{k+1} do
5       if GeneratingParent(f_{k+1}) = f_k then
6         for each g_k ∈ Clones(f_k) do
7           C_{k+1} ← C_{k+1} ∪ {g_k ⊕ e | e.label = f_{k+1}.lastEdge.label}
8         Find(Clones(f_{k+1}))
9         if |Clones(f_{k+1})| < 2 then
10          F_{k+1} ← F_{k+1} \ f_{k+1}
11          Pro ← Pro ∪ f_{k+1}.lastEdge
12      else
13        F_{k+1} ← F_{k+1} \ f_{k+1}
14    for each f_{k+1} ∈ F_{k+1} do
15      L_{k+1} ← L_{k+1} ∪ Clones(f_{k+1})
16      f_{k+1}.Pro ← Pro
17      Discover(f_{k+1}, Clones(f_{k+1}))
```

Listing 5.3: *eScan* discovery avoiding unsuccessful edges

5.2.3. aScan

For the evaluation of ModelCD, we also implemented the *aScan* algorithm. As an intermediate goal of our implementation, we had a version of *aScan* capable of detecting only exact clones. Thus the characteristic vectors of two fragments had to be identical in order for two fragments to be considered in a cloning relationship. Using this simplified version of *aScan*, we could postpone the implementation of a locality sensitive hashing (LSH) function and instead use a normal hash function for clone grouping. However, although we were using all ModelCD and *aScan* optimizations mentioned in [5], our implementation of *aScan* for exact clones failed to process the models MPC and AUT within reasonable time and memory boundaries. Unfortunately, the same scalability issues that hamper the adoption of *eScan* for large models are also valid for *aScan*. In the worst case, *aScan* has to create all subgraphs of the model graph during clone detection. We therefore chose to not continue the evaluation of *aScan* and did not implement the grouping of cloned fragments using LSH.

5.2.4. Clone grouping in ModelCD: Completeness vs. redundancy

Our evaluation of ModelCD supports the finding that ModelCD "gives detection results with (...) much more quantity"[5] compared to CMCD. However, a high quantity of clone groups does not imply high relevance to developers. The developers of our industry partner were confused by clone groups that seemed identical at first glance. In these cases, the overlap criteria caused a clone group to be split into several smaller clone groups as described in Section 3.2.2.2. Consequently, some clone instances are part of multiple clone groups in the output.

In order to avoid a perceived redundancy in the output, we believe that the identification of non-overlapping clone groups should consider the *overlap degree* of fragments.

Figure 5.4 shows a motivating example: Figure 5.4(a) and Figure 5.4(b) depict two non-overlapping subgraphs. They share a big isomorphic core subgraph, denoted by 'A'. The subgraph in Figure 5.4(a) contains one additional node with label x. The subgraph in Figure 5.4(b) contains 4 additional nodes with label x.

eScan initially detects these two subgraphs as a clone group with 5 members $(c_1, c_2, c_3, c_4, c_5)$. c_1 corresponds to the subgraph in Figure 5.4(a). c_{2-5} are composed of the common subgraph 'A' and one of the nodes with label x, depicted in Figure 5.4(b).

During the grouping and filtering phase of ModelCD, the overlap graph in Figure 5.4(c) is constructed. After the maximum clique cover algorithm is executed, ModelCD returns 4 clone groups, each consisting of two clone instances. Each of these clone groups contains c_1 and one of the four clone instances c_{2-5}. The clone report thus contains four groups which differ in only a single node.

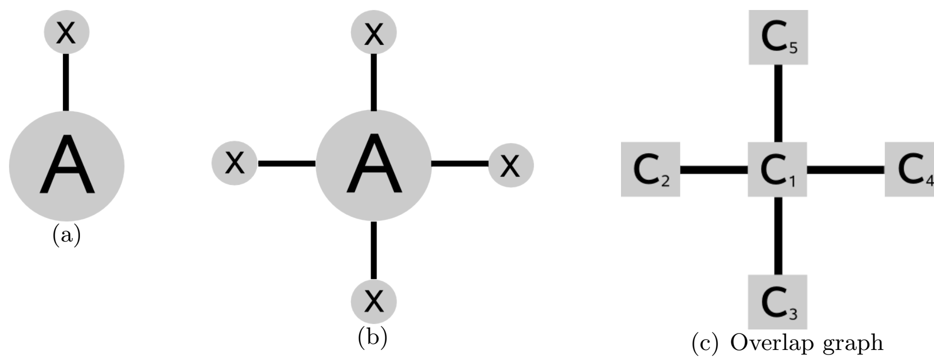


Figure 5.4.: Creation of similar clone groups

While striving to report as many clone groups as possible may be an adequate goal for small models, we believe that a person reviewing the clone detection results of a large model should not be confronted with a high number of clone groups that are very similar to each other. Using an overlap degree of clone instances could help to reduce the number of similar clone groups.

A simple algorithm for removing similar clone groups could work as follows: After the overlap graph has been created, an overlap ratio is calculated for each overlapping pair of clone instances. The overlap ratio is defined as the number of common edges divided by the total number of edges within one clone instance. Each pair of overlapping fragments is then processed in the decreasing order of their similarity values. For each pair with a similarity value above a defined threshold, the node with the lesser degree is removed from the overlap graph. Subsequently, the maximum clique cover of the remaining overlap graph is identified and reported.

5.3. Cloned subsystem detection

We believe that the detection of cloned subsystems in MATLAB/Simulink identifies meaningful clones, as cloned subsystems can easily be externalized into a library element.

This section describes an extended scheme for the detection of cloned subsystems that takes into account the contents of a subsystem.

When ConQAT builds an internal representation of the model graph, all nodes of the Simulink model are visited in a depth first manner. Whenever this DFS backtracks, all nodes within a subsystem have been visited. At this point, a canonical label of the subsystem is calculated and the hash value of the canonical label is used as the subsystem's normalization label. A canonical label is generated as described in Section 3.2.2.3.

During the experiments with the model of our industry partner, we were confronted with few subsystems that contained a high number of nodes with the same normalization label. For those rare cases in which the vertex classification did not result in a complexity reduction (more than 7 ambiguous nodes), we did not calculate all potential adjacency matrices but resorted to a "trivial label" that only consists of the node labels and the edge count with a subsystem.

Of course, this could potentially lead to false positive clones. However, in our case studies no false positives were identified and we believe that in practice, the number of false positives is at least smaller than for the method proposed by Pham *et al.* . If necessary, the false positive rate of our "trivial label" implementation could be improved by including additional information such as the ordered degree sequence into the label.

In our current implementation only those subsystem clones are reported that are not contained within a bigger subsystem clone. For example, in Figure 5.5 only subsystems of type 'A' are reported.

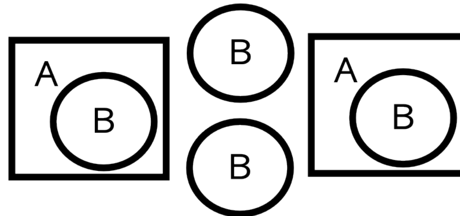


Figure 5.5.: Subsystem B is not reported

One could easily change this implementation so that cloned subsystem groups that are not completely covered by a bigger subsystem group are still reported as cloned subsystems, such as in *eScan*.

5.4. ConQAT's Model Clone Detection

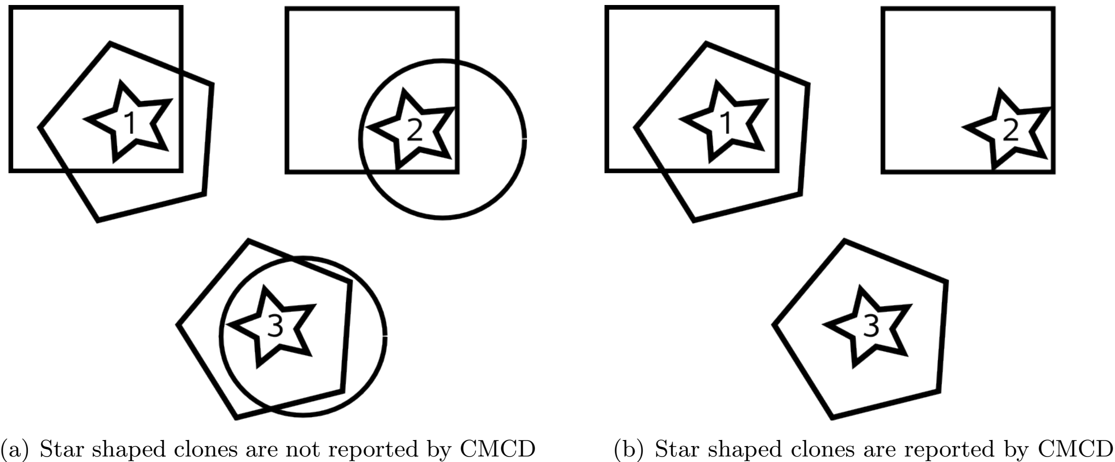
To present a bigger picture on algorithms for graph-based model clone detection we will briefly mention the experiences we gained with ConQAT's model clone detection. During our case study, CMCD scaled very well to large scale models and processed the models of our industry partner within less than an hour on an Intel Pentium 4, 2 GHz, 2 GB RAM. The weaknesses we could identify during our evaluation are mostly those mentioned in the algorithm's original publication [3].

The heuristic approach used by CMCD can have negative impacts on clone detection completeness when "wrong" node pairs are matched. However, as there have been few complaints in practice, we did not to assess this issue for this work.

As CMCD always tries to build maximal clones, smaller clone groups are not reported if for each clone pair, both clones are covered by a bigger clone. In Figure 5.6 geometric figures stand for clone groups. The clones represented by rectangles, circles, and pentagons in Figure 5.6(a) each contain a smaller clone group, represented by a star shape.

CMCD does not discover the star shaped group in Figure 5.6(a) and only reports three clone groups: Starting from nodes in stars 1 and 2 identifies the rectangles, starting from nodes in star 2 and 3 identifies the circles and starting from nodes in stars 1 and 3 identifies the pentagon.

The star shaped group would however be fully discovered if at least one combination of nodes were not covered by a bigger clone. Therefore, the example given by Pham *et al.* for hidden clones not detected by CMCD in [5], depicted in Figure 5.6(b), is incorrect: When using node pairs from star 2 and star 3, CMCD discovers this clone pair. Through an inclusion analysis, CMCD is then able to report the entire star shaped clone group.



(a) Star shaped clones are not reported by CMCD (b) Star shaped clones are reported by CMCD

Figure 5.6.: CMCD hidden clone detection

In the case study of ModelCD, Pham *et al.* claim that CMCD "finds incorrect clones and groups". While this statement is correct *w.r.t.* the definitions used by Pham *et al.* , it should be further qualified as CMCD intentionally uses a slightly different definition of clone groups. By design, CMCD groups clone pairs based on only the node set of a clone. In case this is not desirable, one could of course change this criterion. If desired, overlapping clones within a clone group could be avoided by building an overlap graph and splitting clone groups as it is proposed by ModelCD.

We noticed that the flattening strategy originally used by CMCD endorsed the identification of small clones whose contents were scattered across subsystem boundaries. Although these small clones achieved the minimum required node and weight limits, they were not considered relevant by developers. These small, uninteresting clones were no longer reported when a hierarchical model graph was used.

5.5. gSpan

ParSeMiS [22] is a Java-based graph mining library that contains implementations of a number of graph matching algorithms such as *gSpan*. To check whether *gSpan* is suited for graph-based model clone detection in large-scale models, we used the ParSeMiS implementation of *gSpan* for clone detection in the model MPC. However, for a search in a single-graph environment with a minimum required subgraph frequency of two, the algorithm would not terminate. The algorithm was able to complete its computations when we raised the minimum required pattern frequency to a value above six. As *gSpan* usually works with a higher minimum required frequency, it is not suited for model clone detection in data flow graphs.

5.6. Runtime overview

This section provides an overview of the runtimes required by CMCD and *eScan* to process the models described in Section 5.1. All measurements were performed on a desktop computer running Windows XP with an Intel Pentium 4 CPU with 2 GHz and 2 GB RAM.

Table 5.6 contains the algorithm runtimes we measured, and respectively that were given in the original publication of ModelCD. Durations shorter than one minute are given in seconds. Longer durations are listed in the format mm:ss. We present the algorithms' runtimes for different flattening strategies: For the "Flat structure" strategy, the model graph is flat and contains all model elements but subsystem nodes. The other two strategies use a hierarchical model graph which contains only one instance of cloned subsystems. For "Subsystem name" a subsystem is treated as a clone of another subsystem if both subsystem names are equal. For "Subsystem contents", the approach introduced in this work, a subsystem is regarded as a clone of another subsystem if both subsystem's contents are equal.

For the *eScan* algorithm, we measured the runtimes with and without the high degree node optimization. The row "*eScan* (switch sep.)" contains the runtimes we measured when nodes with a degree above 6 were separated from the model graph at the beginning. The row "*eScan*" lists the algorithm's runtimes without the high degree node optimization.

If a cell contains the entry "-", an algorithm was not able to process a model within the specified memory boundaries in less than 24 hours.

For comparison, the row "*eScan* (publication)" contains the *eScan* runtimes given in its original publication.

Using the original ModelCD optimizations, our *eScan* implementation was capable of processing all models used in the case study performed by Pham *et al.* . Our implementation even performed much faster. This may be due to the new optimizations that we applied or that we calculated canonical labels more efficiently. However, under more demanding circumstances, the algorithm would not terminate for some models. *eScan* could not at all handle the models MPC and AUT.

Algorithm	Flattening strategy	SIM	MUL	SEM	ECW	MPC	AUT
eScan	Flat structure	4.9	-	-	-	-	-
	Subsystem contents	3.8	6.2	-	13:04	-	-
	Subsystem name	4.5	6.8	-	1:05	-	-
eScan (switch sep.)	Flat structure	9.3	-	-	-	-	-
	Subsystem contents	8.3	9.0	-	1:12	-	-
	Subsystem name	6.8	7.3	24.8	37.4	-	-
CMCD	Flat structure	3.4	3.7	13.6	19.5	4.8	54:43
	Subsystem contents	3.3	2.9	7.2	8.3	5.9	31:28
	Subsystem name	3.5	1.9	5.3	4.9	5.6	26:34
eScan (publication)	Subsystem name	2:00	2:12	5:00	10:12		

Table 5.3.: Runtime comparisons

5.7. Summary

Unfortunately, the most important result of our evaluation of algorithms for model clone detection in data flow graphs is that *gSpan* and ModelCD do not scale to the model sizes encountered in practice.

However, ModelCD contains reasonable improvements concerning the treatment of subsystems in Simulink. The detection of cloned subsystems identifies highly relevant clones that can easily be externalized into a library element. Our modified approach offers a fast and more reliable detection of cloned subsystems. If only a single instance of cloned subsystems is included into the model graph, a significant reduction of the model size can be achieved.

A modified version of CMCD now works with hierarchical models and can thus process industry relevant models even faster.

6. Evaluation of clone relevance improvements

This chapter contains our findings on the measures to improve clone relevance proposed in Chapter 4. Section 6.1 assesses the appropriateness of clone filters that use graph distance metrics in clone graphs and Section 6.2 details the outcome of our case study to evaluate the suitability of new clone metrics to rank clones.

6.1. Graph distance filters

When we applied the filters described in Section 4.3 to clones outside the training set, the filters using graph distance metrics, proved to be very unspecific: Many clones were filtered although they would have been interesting and other irrelevant clones were not identified by the filters.

However, the most important shortcoming of our approach was that, apart from the performance of the filters within the training set, we could not justify why the occurrence of a topological pattern identified by our filters would make a clone uninteresting.

The 0-weight filter did not falsely filter any relevant clones, however, it did not filter many irrelevant clones either. Potentially, further research could help to fine-tune this filter to produce better results.

While we can not exclude that the occurrence of specific graph properties does have an impact on the relevance of a clone, we chose not to invest further research in this approach.

6.2. Clone ranking scheme

To assess the validity of our proposed ranking schemes, we performed a case study as described in Section 4.5.

To evaluate the appropriateness of different ranking criteria, we compare the developers' rankings to the rankings gained through a number of clone group metrics. For each clone metric we list the rank deviations from the developers' ranks in Table 6.1. The developers deviated from each other by a value of 74.

Clone group metric	Developer 1	Developer 2
Node size	139	163
Clone group weight	157	183
Modified weight (Interface)	167	178
Modified weight (Rel. weight)	179	198
Interface ranking value	123	123
Rel. weight value	110	90

Table 6.1.: Cumulated rank deviations

Based on the above results, we draw the following conclusions: The developers did not agree upon a shared order of clone groups. This matter is related to reports of code clone detection studies, mentioned in [9], in which developers could rarely agree upon which clones ought to be removed. As a consequence we must accept that there probably is no single perfect ranking. However, this fact does not mean that there are no common criteria which are important to all developers. We should try to identify these criteria.

Furthermore, we can see that the clone group weight metric, which has so far been used to rank clones, scores better than our proposed modified weights. This leads us to the decision not to use the modified weights as ranking criteria. However, a better approximation of the developers' rankings is achieved by using only the interface ranking and the relative weight values as metrics.

After we evaluated the performance of the metrics described above, we tried to identify other metrics that fit the developers' rankings in the case study. Even better results than those described above were achieved by a metric that related a clone group's interface size to its weight. We term this value the *interface weight ratio of a clone group (IWR_{CG})*. The IWR value is computed as the ratio of the average interface size of a group to the clone group weight.

$$IWR_{CG} = \frac{\sum_{c \in CG} c.interfaceSize}{|CG| \cdot W_{CG}}$$

The ranking based on IWR values deviated from the developers by 86 and respectively 90 points and thus comes close to the deviations in between the developers' rankings. However, all clone groups with no interface edges (*e.g.*, cloned subsystems) have a ranking value of zero. These clones could be internally ranked according to their weight.

We are aware of the fact that the validity of this study's results may be limited. The threats to validity are:

- only two developer rankings
- a low number of clones
- the usage of only clone groups with two clone instances
- distortions introduced by the formation of ranks
- results only valid for library identification use case

To put it in a nutshell, due to the limitations of our study we can not constitute the claim that certain metrics offer a better approximation of clone relevance than others. However, we believe that our results justify further research into the validity of the three best performing clone group metrics.

7. Conclusion and future work

This chapter summarizes our findings and gives an overview of potential future work to improve clone detection for models.

7.1. Conclusion

The goal of this thesis was to evaluate existing approaches to graph-based model clone detection and to improve the identified clones' relevance to developers.

As clone detection is, from a problem definition viewpoint, a specialization of graph mining, we assessed the suitability of the graph mining algorithm *gSpan* for clone detection. The implementation of *gSpan* was provided by the well-tested graph mining program library *ParSeMiS*. However, *gSpan* is a complete algorithm that was designed to discover subgraphs occurring within a graph with a much higher frequency than two. This explains why *gSpan* was unable to process relatively small models. Without major adaptations, *gSpan* is therefore not suited for clone detection in graph-based models.

To be able to assess the most recently proposed tool for model clone detection, ModelCD, we implemented the algorithms *eScan* and *aScan* within ConQAT. Unfortunately, our most important finding is that the algorithms do not scale to large models encountered in practice.

Nevertheless, ModelCD can be used within ConQAT as an alternative clone detection strategy that is well suited for medium-sized models. Furthermore, ModelCD introduced reasonable ideas concerning the detection of cloned subsystems. When developers are interested in outsourcing model elements they would like to be notified about cloned subsystems. Externalizing cloned subsystems offers a low cost-benefit ratio as a relatively big model size reduction can be achieved with little effort and a relatively low probability of introducing faults into the system.

As a result of this thesis we have provided a solution that combines the advantages of "both worlds": Our modified cloned subsystem detection provides a quick, meaningful, and reliable reporting on clones that, with a high likelihood, represent a concept in the application domain. Furthermore, a modified version of ConQAT's model clone detection algorithm now works with hierarchical models and can process industry-relevant model sizes within an even shorter amount of time.

While the accurate reporting of cloned subsystems is already a major step to identify highly relevant clones, we analyzed further enhancements to improve the relevance of the reported clones: We concluded that the occurrence of specific topological features within a clone graph does not influence its relevance. We proposed a new ranking scheme for clones *w.r.t.* their appropriateness to be externalized. This ranking scheme uses the interface of a clone, *i.e.*, the edges connecting the clone to the rest of the model, to measure its suitability to form a library. We also evaluated a ranking that calculates a clone's weight *w.r.t.* its size relative to other clones. The ranking schemes we proposed were evaluated in a case

study with our industry partner. We found metrics that better approximated the developers ranking than the clone metrics used beforehand.

In summary, we have identified advantages and drawbacks of existing approaches to graph-based model clone detection, proposed new techniques and identified measures that better estimate a clone's relevance to developers. It can therefore be said that the initial goal was achieved.

7.2. Future work

As the number of model-driven development projects is likely to increase in the future, the need for tools like ConQAT that support the maintainability of models will persist. Therefore, it makes sense to further improve and extend the model clone detection capabilities of ConQAT.

Our newly integrated cloned subsystem detection currently does not report subsystems that are partially covered by bigger cloned subsystems. However, sometimes it is desirable to also report the smaller subsystems. Deciding upon a reporting strategy is a decision in which a tradeoff between detection completeness and a potentially too high number of clones must be made. To strike a balance, a user could define the desired behavior using a configuration parameter. The default value should aim at detection completeness, *i.e.*, report all partially covered cloned subsystems.

Larger studies should be conducted to fully assess the usefulness of the ranking criteria described in this thesis. Potentially, clones with a small interface also indicate relevant clones for other use cases, such as the identification of product line elements. The identified clone metrics could also be used to filter uninteresting clones. Further studies are required for this purpose as the design of our current study does not allow for such conclusions.

A bigger percentage of semantic clones can be identified by the introduction of normal forms for certain constructs. For example, in MATLAB/Simulink, the bitwise AND of three input values can be modeled by a single Bitwise Operator block with three input ports or by two connected Bitwise Operator blocks with two input ports each. Although both ways model the same idea, current clone detection algorithms would not consider these fragments as clones because they are not isomorphic. However, if a variety of modeling possibilities is transformed into a single standard representation, a higher number of clones can be identified by current clone detection algorithms.

Further metrics could be used to rank the relevance of clones. For code clone detection it has been suggested that clones should be ranked higher if the clone instances are situated in different source files - the rationale being that the local separation indicates an unintentional duplication of code. Although this claim has not been verified for code clone detection, it could be that model clones occurring in different model files are more relevant to developers.

Future rankings could also take into account facts about the application domain. Working with our automotive industry partner revealed interesting heuristics: The developers were particularly interested in clone groups containing four clone instances as this oftentimes indicated that the same computation was modeled for each of a car's wheels.

The relevance of clones could be improved by the removal of uninteresting parts: For example, one could remove those nodes within a clone graph that have a weight of 0, if their removal does not disconnect the clone graph. The heuristics could also take into

account semantics of Simulink: For example, if a clone contains an *If*-Block but none of the corresponding statements, the *If*-Block can be removed from the clone.

There is still potential for algorithmic improvements: While ConQAT's Model Clone Detection is scalable, its heuristic design can lead to incomplete results. On the other hand, most graph mining algorithms are complete but do not scale for the purpose of clone detection in large models. Potentially, new algorithms could already consider a clone's weight during the detection phase.

To better assess the relevance of clones, we hope to get feedback from users that apply ConQAT for model clone detection. Hopefully, this will result in new and promising solutions to ease the maintenance of models.

Appendix

A. Detailed case study results

This chapter contains detailed information about the case study we describe in Section 4.5. Table A.1 contains for all clones the raw values for all graph metrics that we assessed. See Section 4.5 and Section 6.2 for a definition of the notation. For metrics that are sensitive to a block weight assignment, the assignment scheme is given in brackets. AUT stands for the weighting scheme of our industry partner, Def refers to ConQAT's default Simulink weights.

Clone ID	# nodes	W_{CG} (AUT)	$MW_{CG}^{RV^i}$ (AUT)	$MW_{CG}^{RV^w}$ (AUT)	W_{CG} (Def)	$MW_{CG}^{RV^i}$ (Def)	$MW_{CG}^{RV^w}$ (Def)	RV_{CG}^i	RV_{CG}^w (AUT)	RV_{CG}^w (Def)	IWR_{CG} (AUT)	IWR_{CG} (Def)
1	18	142	138.06	112.02	32	31.11	28.44	0.97	0.79	0.89	0.01	0.03
2	10	20	12.06	4.00	4	2.40	0.80	0.60	0.20	0.20	0.40	2.00
3	17	22	19.29	2.85	8	7.01	1.88	0.88	0.13	0.24	0.18	0.50
4	27	204	196.40	154.13	54	51.99	54.00	0.96	0.76	1.00	0.01	0.04
5	105	208	195.10	41.20	106	99.43	53.50	0.94	0.20	0.50	0.06	0.12
6	14	100	96.40	71.43	28	26.99	28.00	0.96	0.71	1.00	0.01	0.04
7	15	102	0.00	69.36	28	0.00	26.13	0.00	0.68	0.93	0.29	1.07
8	7	30	0.00	12.86	6	0.00	2.57	0.00	0.43	0.43	0.47	2.33
9	15	150	20.00	150.00	30	4.00	30.00	0.13	1.00	1.00	0.17	0.87
10	18	32	1.78	5.69	16	0.89	7.11	0.06	0.18	0.44	1.06	2.13
11	11	110	0.00	110.00	20	0.00	18.18	0.00	1.00	0.91	0.20	1.10
12	8	62	54.25	48.05	12	10.50	9.00	0.88	0.78	0.75	0.03	0.17
13	12	40	10.00	13.33	20	5.00	16.67	0.25	0.33	0.83	0.45	0.90
14	23	206	170.17	184.50	46	38.00	46.00	0.83	0.90	1.00	0.04	0.17
15	18	154	111.20	131.76	32	23.11	28.44	0.72	0.86	0.89	0.06	0.31
16	8	64	36.00	51.20	10	5.63	6.25	0.56	0.80	0.63	0.11	0.70
17	7	38	5.43	20.63	10	1.43	7.14	0.14	1.54	0.71	0.16	0.60
18	23	148	77.20	95.23	40	20.86	34.78	0.52	0.65	0.87	0.15	0.55
19	21	210	40.00	210.00	42	8.00	42.00	0.19	1.00	1.00	0.16	0.81
20	9	68	3.78	51.38	16	0.89	14.22	0.06	0.76	0.89	0.25	1.06
21	14	104	89.14	77.26	28	24.00	28.00	0.86	0.74	1.00	0.04	0.14

Table A.1.: Raw graph metric values for clones used in the case study

Table A.2 contains for all clones the ranks that a clone received under all graph metrics and respectively by the developers. At the bottom, for each metric, the deviations $D_{d,m}$ for both developers are shown. For all metrics, except $IWR_{CG}(AUT)$ and $IWR_{CG}(Def)$, fractional ranks are assigned in a descending order of their raw values. The ranks for $IWR_{CG}(AUT)$ and $IWR_{CG}(Def)$ are assigned in an ascending order as for this metric a "good" clone has a small value.

Clone ID	Developer 1	Developer 2	# nodes	W_{CG} (AUT)	$MW_{CG}^{RV^i}$ (AUT)	$MW_{CG}^{RV^w}$ (AUT)	W_{CG} (Def)	$MW_{CG}^{RV^i}$ (Def)	$MW_{CG}^{RV^w}$ (Def)	RV_{CG}^i	RV_{CG}^w (AUT)	RV_{CG}^w (Def)	IWR_{CG} (AUT)	IWR_{CG} (Def)
1	4.5	3	15	14	18	16	15.5	18	14	1	7	10	1	1
2	20	17	6	1	8	2	1	7	1	10	18	21	18	19
3	16.5	17	13	2	9	1	3	11	2	5	21	20	14	9
4	2	1	20	18	21	19	20	20	21	3	9	3.5	2	3
5	20	17	21	20	20	7	21	21	20	4	19	17	7	4
6	10	10.5	9.5	10	16	12	12	17	12	2	12	3.5	3	2
7	8	17	11.5	11	2	11	12	2	11	20	13	7	17	17
8	1	10.5	1.5	3	2	4	2	2	3	20	16	19	20	21
9	10	8	11.5	16	10	18	14	8	16	16	1	3.5	13	14
10	16.5	17	15	4	4	3	7.5	5	5	17	20	18	21	20
11	16.5	5	7	13	2	15	9.5	2	10	20	1	8	15	18
12	6	7	3.5	7	13	8	6	13	7	6	8	14	4	6
13	20	17	8	6	7	5	9.5	9	9	13	17	13	19	15
14	3	4	18.5	19	19	20	19	19	19	8	4	3.5	6	7
15	14	17	15	17	17	17	15.5	15	14	9	5	10	8	8
16	7	17	3.5	8	11	9	4.5	10	4	11	6	16	9	12
17	12.5	10.5	1.5	5	6	6	4.5	6	6	15	15	15	11	11
18	12.5	10.5	18.5	15	14	14	17	14	17	12	14	12	10	10
19	10	2	17	21	12	21	18	12	18	14	1	3.5	12	13
20	16.5	17	5	9	5	10	7.5	4	8	18	9	10	16	16
21	4.5	6	9.5	12	15	13	12	16	12	7	11	3.5	5	5
D_1	0	74	139	157	167	179	157	165	159	123	110	102	86	103
D_2	74	0	163	183	178	198	182	178	186	123	90	87	90	102
$\sum_{x=1}^{2} D_x$	74	74	302	340	345	377	339	343	345	246	200	189	176	205

Table A.2.: Ranks assigned for clones under their respective ranking

Bibliography

[1] Anne Angermann, Michael Beuschel, and Martin Rau. *Matlab - Simulink - Stateflow. Grundlagen, Toolboxen, Beispiele*. Oldenbourg, 2007.

[2] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 603–612, New York, NY, USA, 2008. ACM.

[4] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 440–455, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 276–286, Washington, DC, USA, 2009. IEEE Computer Society.

[6] Fortin S. The graph isomorphism problem. Technical report tr96-20, Department of Computing Science, University of Alberta, Edmonton, Canada, 1996.

[7] Reinhard Diestel. *Graph Theory*. Springer-Verlag, Heidelberg, 2005.

[8] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[9] Rainer Koschke. Survey of research on software clones. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[10] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[11] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[12] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

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

[14] Diane J. Cook and Lawrence B. Holder. *Mining Graph Data.* John Wiley & Sons, 2006.

[15] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[16] Michihiro Kuramochi and George Karypis. Finding frequent patterns in a large sparse graph. *Data Min. Knowl. Discov.*, 11(3):243–271, 2005.

[17] Ronald C. Read and Derek G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363, 1977.

[18] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[19] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining*, page 721, Washington, DC, USA, 2002. IEEE Computer Society.

[20] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 1 edition, July 2008.

[21] Fred Buckley and Frank Harary. *Distance in Graphs.* Perseus Books, 1990.

[22] Marc Wörlein, Thorsten Meinl, Ingrid Fischer, and Michael Philippsen. A quantitative comparison of the subgraph miners mofa, gspan, ffsm, and gaston. In *PKDD*, pages 392–403, 2005.