Technische Universität Braunschweig

Bachelor's Thesis

Incremental Construction of Modal Implication Graphs for Feature-Model Evolution

Author:

Rahel Arens

January 08, 2021

Advisors:

Prof. Dr.-Ing. Ina Schaefer

Michael Nieke, M.Sc.

Institute of Software Engineering and Automotive Informatics
TU Braunschweig

Prof. Dr.-Ing. Thomas Thüm

Institute of Software Engineering and Programming Languages
Ulm University

ISF Institut für Softwaretechnik und Fahrzeuginformatik

https://doi.org/10.24355/dbbs.084-202107050823-0

Arens, Rahel:
Incremental Construction of Modal Implication Graphs for Feature-Model Evolution
Bachelor's Thesis, TU Braunschweig, 2021.

https://doi.org/10.24355/dbbs.084-202107050823-0

Abstract

Management of configurable software is often supported by using software product lines. A feature model represents a software product line by modeling the configurable features in a hierarchical tree-structure. Additionally, it defines the set of valid configurations. When constructing a configuration, a SAT solver is called multiple times after each (de-)selection of a feature to determine which features have to be (de-)selected afterwards. For a large-scale feature model, the high number of necessary SAT calls may take up to several hours. A modal implication graph (MIG) describes dependencies between features to reduce the number of SAT calls in the configuration process and, thus, the required time. However, the construction of a MIG can take up to several hours for large feature models. Moreover, the MIG has to be recalculated after every change in a feature model with state-of-the-art methods which leads to high computational effort. In this thesis, we introduce a concept to incrementally calculate a MIG by adapting it to changes in the feature model. To this end, we calculate and use the differences in the conjunctive normal forms (CNFs) representing the feature model versions before and after an evolution step. Based on these CNF differences, the MIG is modified. We evaluate our concept with regard to (1) the correctness of the incremental calculation, (2) the advantage in the computation time when incrementally calculating, (3) the number of SAT calls we reduced, and (4) the accuracy of the incremental calculation. Our empirical evaluation strongly indicates high benefits of the incremental construction regarding the number of required SAT calls and computation time. Our empirical evaluation shows that the incremental construction saves up to 91% of computation time and up to 86% of SAT calls.

https://doi.org/10.24355/dbbs.084-202107050823-0

https://doi.org/10.24355/dbbs.084-202107050823-0

Inhaltsangabe

Die Verwaltung von konfigurierbarer Software wird häufig durch die Verwendung von Software-Produktlinien unterstützt. Ein Feature-Model repräsentiert eine Software-Produktlinie durch Modellierung der konfigurierbaren Features in einer hierarchischen Baumstruktur. Zusätzlich definiert es die Menge der gültigen Konfigurationen. Beim Erstellen einer Konfiguration wird ein SAT-Solver nach jeder (De-)Selektion eines Features mehrfach aufgerufen, um zu bestimmen, welche Features danach (de-)selektiert werden müssen. Die hohe Anzahl der notwendigen SAT-Aufrufe kann für komplexe Feature-Modelle zu Laufzeiten von mehreren Stunden führen. Ein modaler Implikationsgraph (MIG) beschreibt Abhängigkeiten zwischen Features, um die Anzahl der SAT-Aufrufe im Konfigurationsprozess und damit die benötigte Zeit zu reduzieren. Allerdings kann die Konstruktion eines MIGs bei großen Feature-Modellen bis zu mehrere Stunden dauern. Außerdem muss der MIG nach jeder Änderung eines Feature-Models bei aktuellem Stand der Technik neu berechnet werden, was zu einem hohen Rechenaufwand führt. In dieser Arbeit führen wir ein Konzept zur inkrementellen Berechnung eines MIGs ein, indem der MIG nach einem Evolutionsschritt angepasst wird statt neu berechnet zu werden. Zu diesem Zweck berechnen und nutzen wir die Unterschiede in den konjunktiven Normalformen (CNFs), die die Feature-Model Versionen vor und nach einem Evolutionsschritt repräsentieren. Basierend auf dieser CNF-Differenz wird der MIG angepasst. Wir evaluieren unser Konzept im Hinblick auf (1) die Korrektheit der inkrementellen Berechnung, (2) den Vorteil in der Rechenzeit bei inkrementeller Berechnung, (3) die Anzahl der reduzierten SAT-Aufrufe und (4) die Genauigkeit der inkrementellen Berechnung. Unsere empirische Evaluation zeigt signifikante Vorteile der inkrementellen Konstruktion in Bezug auf die Anzahl der benötigten SAT-Aufrufe und die benötigte Berechnungszeit. Unsere empirische Evaluation zeigt, dass die inkrementelle Konstruktion bis zu 91% der Berechnungszeit und bis zu 86% der SAT-Aufrufe einspart.

https://doi.org/10.24355/dbbs.084-202107050823-0

Contents

https://doi.org/10.24355/dbbs.084-202107050823-0

https://doi.org/10.24355/dbbs.084-202107050823-0

List of Figures

https://doi.org/10.24355/dbbs.084-202107050823-0

List of Tables

1. Introduction

Managing configurable software systems is a complex task as they may contain a high number of features with a much higher number of different variants [BSRC10, IF10, KTS+18, STS20, KZK10]. To support the development of configurable systems a product line (PL) can be defined to manage the features and facilitate reusability of their functionalities. With a feature model (FM), the different features of a PL and the dependencies between the features can be organized [BSRC10, SSK+20, Bat05, CHE05, CW07]. The FM represents the features in a hierarchical tree structure with cross-tree constraints for the dependencies that cannot be expressed by the tree [BSRC10, KTS+18]. A configuration is a combination of selected features for a product. We call a configuration valid if its selected feature do not violate any of the constraints that are imposed by the feature model and a FM valid if there is at least one valid configuration for the FM [BSRC10, KTS+18]. To analyze an FM, e.g. to verify whether it is valid, it is typically translated into a conjunctive normal form (CNF) which can be verified by a SAT solver [Bat05, BSRC10]. A SAT solver is a tool to check whether a given boolean formula is satisfiable. Using a SAT solver, we can also verify the validity of a given configuration [KTS+18]. When constructing a configuration, it is important to keep the user informed what features are left to be selected or have to be selected now. To this end, a SAT solver needs to be called several times after each selection or deselection of a feature. Using the unfinished configuration, the SAT solver can be used to determine the remaining possible selections. That process is called decision propagation [KTS+18]. For large-scale FMs, decision propagation requires high numbers of SAT calls, each potentially taking up to several hours [KTS+18].

A Modal Implication Graph (MIG) is a directed graph that was introduced to reduce the number of SAT calls necessary for the decision propagation. A MIG describes the dependencies between variables of a propositional formula [KTS+18]. Instead of calling a SAT solver several times after a (de-) selection of a feature, we can use a MIG to support the decision propagation regarding the computation time. Each node in a MIG represents a selected or deselected feature. The nodes can be connected by edges. We distinguish between the construction of a MIG and the usage of the MIG afterwards. A MIG is calculated once after the construction of a

https://doi.org/10.24355/dbbs.084-202107050823-0

FM. Calculating a MIG from scratch requires a lot of time [KTS+18]. But since this step is done without any user interaction and improves the user experience in the configuration construction, the MIG brings an added value. After the construction it can be used until the respective FM is changed. In this case, the MIG is no longer valid, even when the changes in the FM are small and the main part of the MIG still is valid. Recalculating the MIG can take up to several hours, depending on the size of the FM.

To improve the usage of MIGs in the case of FM evolution, we introduce a method to adapt an existing MIG after changes in a FM. This way, we aim to accelerate the construction of the MIG regarding the necessary calculation time when calculating it from scratch. Our main focus is to reduce the number of required SAT calls when constructing the MIG. For the construction of a MIG, the FM's CNF is used as basis of the calculation. We also use the CNF for the incremental construction but we use the changes in the CNFs of the old and the new FM as basis. The from scratch calculated MIG is improved by multiple steps to make the MIG smaller on the one hand and more efficient on the other hand. It is possible to skip some of the additional steps and still have a resulting MIG that is correct. In this case we call the MIG incomplete. Otherwise, we call the MIG complete. For the incremental construction of a MIG, we also distinguish between an incomplete and a complete MIG.

In summary, we provide concepts and algorithms for the following steps to incrementally compute MIGs:

1. Compute differences between two FM versions in terms of differences of their respective CNFs
2. Incrementally compute incomplete MIGs
3. Incrementally compute complete MIGs

We implement the algorithms as addition to the already existing calculation of a MIG in the feature modeling toolsuite FeatureIDE [TKB+14]. We empirically evaluate the incremental adaption with different feature models that have a different number of changes, features, and constraints. To this end, we measure the calculation times and the number of SAT calls needed for the calculation from scratch and the incremental construction. Additionally, we investigate the difference between the two resulting MIGs and the trade-offs between runtime and accuracy of the incremental construction. Our evaluation shows saving of up to 91% of computation time and up to 86% of SAT calls while keeping a high degree of accuracy.

We provide the basics that are necessary to understand the rest of the thesis in Chapter 2. Afterwards, we introduce multiple new algorithms for the incremental construction in Chapter 3. In Chapter 4, we explain the implementation of our algorithms in FeatureIDE, the additional challenges we had to cope with to implement the incremental construction, and the adaptions that were necessary in the existing MIG for the incrementally constructed MIG. In Chapter 5 we present the results of our empirical evaluation. We discuss related work in Chapter 6. Finally, in Chapter 7, we provide a conclusion and discuss future work on the incremental construction.

2. Background

In this chapter, we give an overview of the knowledge and techniques that are the basis of this thesis. To this end, we explain feature models and their semantics, the analysis of feature models using propositional logic with an emphasis on CNFs and SAT solvers, and the concept of a modal implication graph.

2.1 Feature Models

A *software product line* (SPL) manages the different functionalities of a variable software system. A software product line contains variable artifacts that support reusing code of features in different products. A variable software system typically consists of multiple distinct fragments called *features* which can be composed to derive a variety of products. Constructing multiple products without a product line leads to repeated implementations. A *feature model* (FM) can be used to describe a software product line on an abstract level [Bat05, CHE05, CW07]. It contains all features of the SPL and defines all valid combinations of features. For a graphical representation of a FM, a feature diagram visualizes the features and the relationships among them. In a feature diagram, the features are displayed hierarchically in a tree structure. If a selection of a feature always leads to the selection of a child features, we call this child feature *mandatory*. Otherwise, we call it *optional*. Dependencies that cannot be represented by the tree structure are typically displayed separately as propositional constraints (i.e., cross-tree constraints).

Figure 2.1 shows an example of a feature diagram for an election registration software. The FM contains two mandatory features: Way of voting and Identification. When selecting the way of voting, the features below are part of an *alternative* group. This means, the voter can choose either voting in person or by post. When choosing how to identify, the features below are part of an *or* group. That means, the voter can identify either by using an online service, at a polling station, or both. Optionally, the voter can specify the city he is from.

A combination of features that can be used to derive a product is called a *configuration* [BSRC10]. We call a configuration *valid* if the chosen combination of features

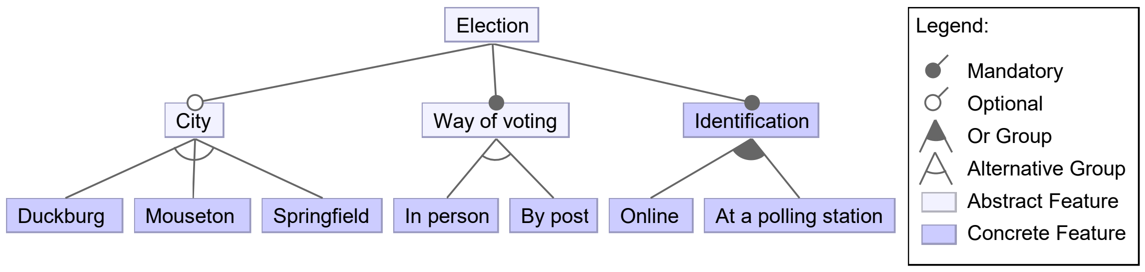


Figure 2.1: Example of a feature diagram

does not violate any constraint imposed by the FM (e.g., Election, Way of voting, In person, Identification, Online). If there exists no valid configuration for a FM, we call the FM *void*. We refer to features that have to be selected for a valid configuration as *core*. In our example, the features Election, Way of voting, and Identification are core features. As a FM grows, different constraints may lead to features that have to be deselected in each valid configuration [KZK10, STS20]. We refer to these features as *dead*. In the following, we explain how to verify the validity of a configuration with regards to FM constraints.

2.2 Feature Models and Logics

A *conjunctive normal form* $CNF = \{L, C\}$ is a propositional formula that consists of a set of literals $L = (l_1, \neg l_1, .., l_i, \neg l_i)$ and a set of clauses $C = (c_1, .., c_n)$. A literal corresponds to a single variable and can either be positive or negative. A single clause represents a dependency between literals, where the literals within the clause are connected with a logical or (i.e., $c_1 = \{l_j, .., l_m\} = l_j \vee .. \vee l_m$). The clauses contained in the CNF are connected with a logical and (i.e., $c_1 \wedge .. \wedge c_n$). We call a specific selection and deselection of features an *assignment* (e.g., $l_3 \wedge l_7 \wedge \neg l_9$). An assignment satisfies a CNF if it does not violate any of the clauses of the CNF. A CNF is satisfiable if there is a valid assignment for the CNF. If the CNF represents a FM, the literals describe either a selected or a deselected feature. We translate a FM into a CNF to check if the FM is not void (i.e., if the CNF is satisfiable). Every existing FM can be translated into a CNF [Bat05]. The corresponding CNF for Figure 2.1 is the following:

$$\begin{gathered}
CNF = \big(\{Election,\ City,\ Way\ of\ voting,\ Identification,\ Duckburg,\ Mouseton,\\
Springfield,\ In\ person,\ By\ post,\ Online,\ At\ a\ polling\ station\},\ \{\ \{Election\},\\
\{Way\ of\ voting\},\ \{Identification\},\ \{Duckburg,\ Mouseton,\ Springfield\},\\
\{\neg Duckburg,\ \neg Mouseton\},\{\neg Springfield,\ \neg Mouseton\},\\
\{\neg Duckburg,\ \neg Springfield\},\ \{City,\ \neg Mouseton\},\ \{City,\ \neg Duckburg\},\\
\{City,\ \neg Springfield\},\ \{\neg City,\ Mouseton,\ Springfield,\ Duckburg\},\\
\{\neg In\ person,\ \neg By\ post\},\ \{In\ person,\ By\ post\},\ \{Online,\ At\ a\ polling\ station\}\ \}\ \big)
\end{gathered}$$

We can see three kinds of clauses in the CNF: (1) clauses with one feature, called *unit clauses*, (2) clauses with two features that have a binary relation, and (3) clauses with three features. A unit clause, e.g. $\{Election\}$, indicates that the contained feature has to be selected for a valid configuration. Having a negated feature in a unit clause means that it has to be deselected. A clause with two features means that at least one of these features has to be selected or, if negated, deselected. For example, the clause $\{\neg$*In person,* $\neg$*By post*$\}$ means that when selecting In person, By post has to be deselected. In combination with the clause $\{$*In person, By post*$\}$ it expresses that either In person or By post has to be selected but not both.

To check whether a specific feature is core or dead, we add the corresponding literal (i.e., a positive literal for core and a negative literal for dead) as a unit clause to the CNF. Afterwards, we check if the resulting CNF is still satisfiable. If it is not, the corresponding feature is dead if the literal in the unit clause is positive and core otherwise. In the following, we explain the usage of SAT solvers to verify the satisfiability of a CNF.

A SAT solver is a tool that checks whether a boolean formula is satisfiable [Jan08]. We can use a SAT solver to verify if a CNF induces any satisfying assignments. To this end, the SAT solver gets a CNF as input and returns true if there exists an assignment for the CNF. If not, it returns false. We can check if a FM is valid by giving the FM's CNF as input to a SAT solver. Additionally, a SAT solver can verify the validity of a given configuration. When selecting a feature, we add the corresponding literal as unit clause to the CNF of the FM and when deselecting a feature, we add the corresponding negated literal as a unit clause to the CNF. Then, we verify with a SAT solver if the resulting CNF is still satisfiable. A SAT solver can also be used to check whether any other feature has to be selected or deselected after selecting or deselecting a specific feature. For example, when selecting the feature Duckburg we can add the unit clauses $\{$*Duckburg*$\}$ and $\{$*Mouseton*$\}$ to the SAT solver. The SAT solver would return false and, thus, we know that after selecting Duckburg, Mouseton can not be selected anymore. When a user constructs a configuration, we can keep him informed after each selection or deselection of a feature which features have to be selected or deselected now. This is called *decision propagation.* For large-scale feature models, the high number of SAT calls necessary to provide decision propagation cost several minutes for each decision in the configuration process [KTS+18]. In the following, we explain modal implication graphs that were introduced to reduce the number of SAT calls necessary for decision propagation [Kri15].

2.3 Modal Implication Graphs

A *modal implication graph* (MIG) is a directed graph $G = (L, E)$ with a set of literals L as nodes and a set of edges E consisting of *weak* and *strong* edges [KTS+18, Kri15]. A MIG represents the dependencies between features of a FM. Figure 2.2 shows the corresponding MIG for the FM example in Figure 2.1. A strong edge represents a binary relation between two literals and is displayed in the graph by a black triangle arrow $\twoheadrightarrow$. Let f be a feature, l_f a positive literal corresponding to f, and $l_f \twoheadrightarrow l_g$ and $l_f \twoheadrightarrow \neg l_h$ be strong edges to literals corresponding to the features g and h. If f has been selected, g has to be selected too. Additionally, h has to be deselected. For instance, when selecting the feature Springfield, the strong edges going away from the corresponding literal show that Mouseton and Duckburg have to be deselected now because strong edges go from Springfield to $\neg$Mouseton and to $\neg$Duckburg. A weak edge represents a connection between two literals that are part of an n-ary clause with at least three literals. In the MIG, a weak edge is displayed by a white triangle arrow $\rightarrow\!\triangleright$. For example, when deselecting the feature Duckburg, the MIG shows that either Mouseton or Springfield have to be selected now.

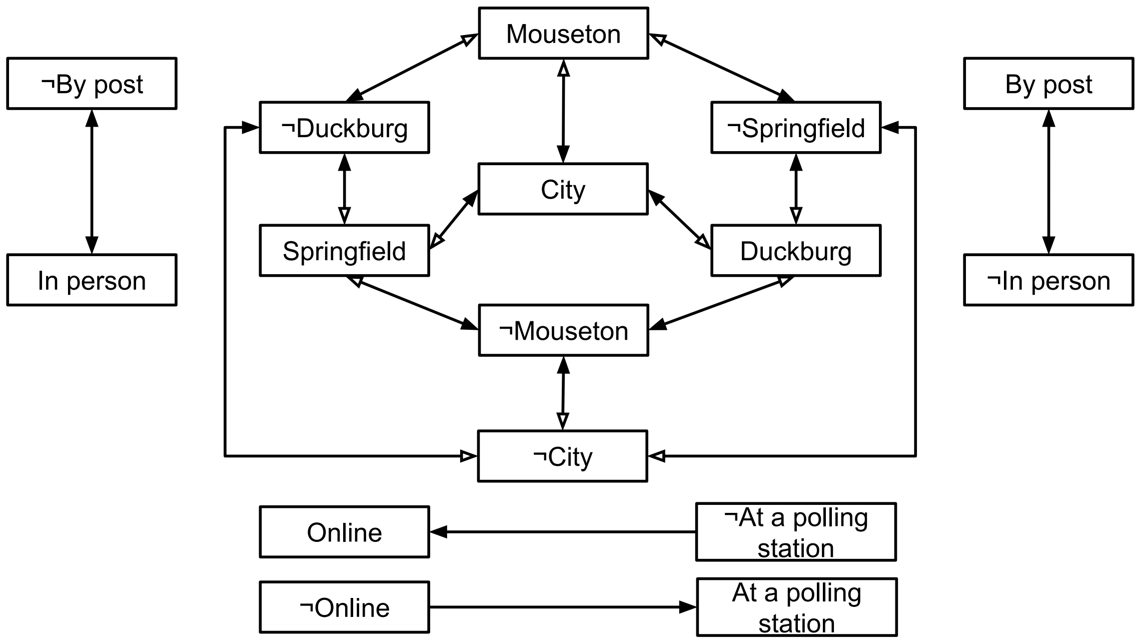


Figure 2.2: Example of a modal implication graph

The literature distinguishes between incomplete and complete MIGs. The construction of an incomplete MIG includes three steps.

1. The first step is to collect all clauses that have to be added to the MIG. That includes finding core and dead features and removing clauses from that CNF that are either redundant (i.e., their logic is already defined by a set of other clauses) or a tautology (i.e., the clause is satisfied for every possible assignment). For redundant clauses, consider the following example. We have a CNF = $\{ \{A,\ B\},\ \{A,\ B,\ C\}\}$. The clause $\{A,\ B,\ C\}$ is a redundant clause, because the clause $\{A,\ B\}$ already defines that either A or B have to be selected. For a tautology, consider the clause $\{A,\ \neg A,\ B\}$. This clause is satisfied if A is either selected or deselected and, thus, always satisfied.

2. The second step is to add all of these clauses to the MIG. For each variable appearing in the CNF, we add two vertices corresponding to the positive and negative literals to the MIG. For core and dead features, we add no edge to the MIG but save the information for the corresponding literals. We do so, because the MIG displays the connection between features and to be core or dead is an attribute of a feature itself. Strong edges are added for clauses with two literals, e.g., for the clause $\{\neg Duckburg,\ \neg Mouseton\}$ in the CNF representing our running example, we add a strong edge from Duckburg to ¬Mouseton and a strong edge from Mouseton to ¬Duckburg. For clauses with

three or more literals, we add two weak edges for each pair of variables. For example, for the clause {*Duckburg, Mouseton, Springfield*} we add a weak edge from ¬Duckburg to Mouseton and from ¬Mouseton to Duckburg. We add weak edges analogously for the other pairs.

3. Afterwards, transitive strong edges are calculated. For our example MIG in Figure 2.2, consider an additional strong edge from ¬Duckburg to Mouseton. Since the MIG also contains a strong edge from Springfield to ¬Duckburg, selecting Springfield always requires to select Mouseton. Thus, we add a transitive strong edge from Springfield to Mouseton.

Transforming an incomplete MIG to a complete MIG takes two additional steps.

1. The first one is finding transitive weak edges and adding them to the MIG. This works similar to the way that transitive strong edges were detected in the construction of the incomplete MIG.

2. The second step is detecting implicit strong edges. These edges are weak edges that can be transformed to strong edges. The reason for the possible transition can be caused due to other constraints and edges. Consider the following example. We have the clauses $\{A, B, C\}$ and $\{\neg A\}$. The weak edges from $\neg B$ to C and from $\neg C$ to B are implicit strong edges, since A can never be selected and, thus, either B or C has to be selected.

A MIG represents a lot of information about the dependencies among features which simplifies the construction of a valid configuration. When using a MIG for decision propagation, several SAT calls can be saved as the MIG provides knowledge about the consequences that result from the selection or deselection of features.

3. Updating Modal Implication Graphs

During the evolution of a variable system, the corresponding FM typically changes as well. After a new change to the FM, the MIG has to be adapted to the changes. Algorithms considered in the literature require a construction of the corresponding MIG from scratch. Building a new MIG for each new FM version typically costs a lot of computation time [KTS+18]. We introduce a method to adapt an existing MIG to changes in the FM. The goal of incrementally constructing a MIG is to reduce the time required for the MIG calculation after evolution steps. When constructing a MIG, the main computational effort lies in invocations of SAT solvers. Therefore, our main focus lies on reducing the required SAT queries with the incremental construction.

In this chapter, we highlight challenges that we identified for incrementally creating a MIG. We explain the concepts of our method including algorithms to solve the aforementioned challenges. When building a MIG for an FM, the FM is translated into a CNF. Subsequently, the clauses of the CNF are used to build the MIG (cf. Section 2.3). Instead of using the FM for incremental construction, we use the changes in the CNF. As Figure 3.1 shows, we identified calculating the difference between the CNFs before and after FM evolution as the first step. Calculating the incomplete MIG is the first step when constructing a MIG. Afterwards, a complete MIG is built with the incomplete MIG as its base. To introduce our method, we differentiate between incomplete and complete MIGs as introduced in Section 2.3.

To demonstrate the CNF difference resulting from changes to its FM, consider the feature diagram from Figure 2.1 and add the cross-tree constraint By post $\Rightarrow$ Duckburg $\vee$ Mouseton (cf. Figure 3.2). Introducing the constraint adds the clause $\{\neg$*By post, Duckburg, Mouseton*$\}$ to the CNF. Thus, when calculating the CNF difference, we detect exactly this clause as a newly added clause.

The method `calculateCnfDiff` (cf. Algorithm 1) shows how we calculate the difference between the CNF representing the old FM and the CNF representing the

https://doi.org/10.24355/dbbs.084-202107050823-0

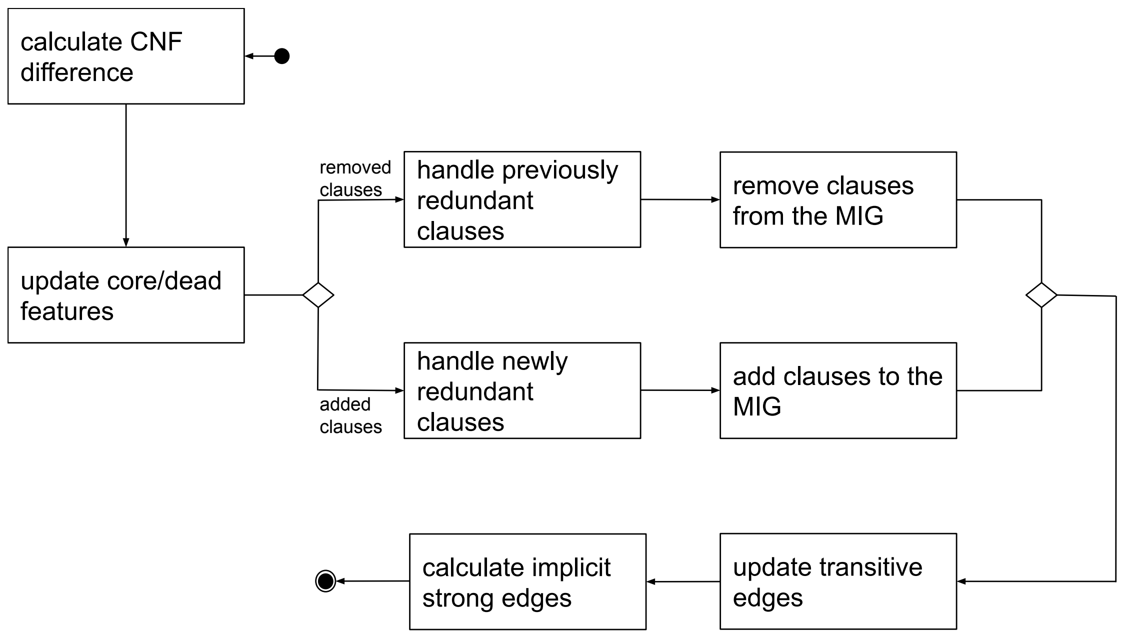

calculate CNF difference
update core/dead features
removed clauses
added clauses
handle previously redundant clauses
handle newly redundant clauses
remove clauses from the MIG
add clauses to the MIG
update transitive edges
calculate implicit strong edges

Figure 3.1: Order of events when updating the MIG

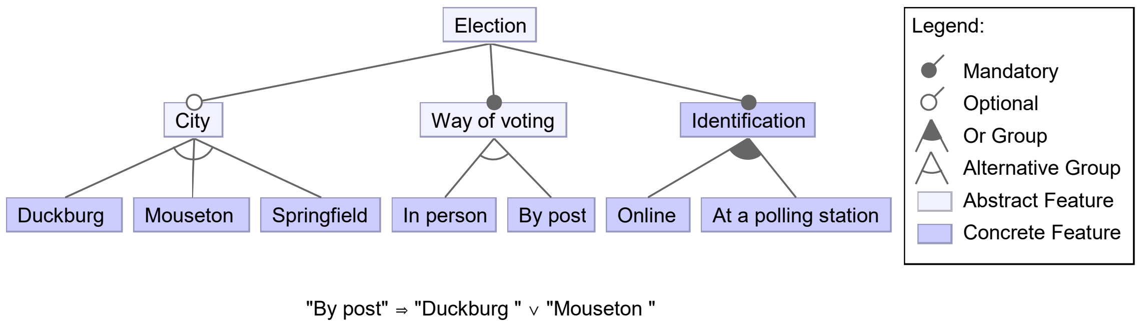


Figure 3.2: Example of a feature diagram after an evolution step

changed FM. We differentiate between removed and added clauses. For instance, adding a new constraint in the FM may result in an additional literal in an already existing clause. We define the difference of two CNFs as the sets of removed and added clauses. To retrieve the removed clauses, we remove all clauses of the new CNF from the old CNF in lines 2 and 3. For the added clauses, we remove all clauses from the new CNF that are in the old CNF in lines 4 and 5.

The next step, as shown in Figure 3.1, is calculating dead and core features, as they might have changed because of the changes in the FM. To not detect these changes would lead to a wrong MIG. Additionally, knowing which features are dead and core simplifies detecting redundant clauses. Changes to the CNF may have an impact on the redundancy of clauses. Removing a clause may affect other clauses to lose their redundancy. Adding a clause may make other clauses redundant that were not redundant before. We discuss the details of handling redundant clauses when incrementally constructing a MIG in Section 3.1.1. Afterwards, we remove and add the clauses that we detected by calculating the CNF difference. Finally,

Algorithm 1 Algorithm to calculate the difference between two CNFs

```
1: function calculateCnfDiff(cnfBeforeChanges, cnfAfterChanges)
2:     removed ← cnfBeforeChanges.getAllClauses()
3:     removed ← removed.removeAll(cnfAfterChanges.getClauses())
4:     added ← cnfAfterChanges.getAllClauses()
5:     added ← added.removeAll(cnfBeforeChanges.getClauses())
   return (removed, added)
```

we update transitive edges in the MIG. For the incomplete MIG, we only update strong transitive edges. To this end, we find literals without a strong edge that are connected over at least two strong edges. For the complete MIG, we update weak transitive edges (i.e. literals that are not connected that have a relation over at least two other edges including at least one weak edge) and calculate implicit strong edges (i.e., weak edges that can be updated to strong edges due to the impact of other clauses).

3.1 Incomplete Modal Implication Graphs

Our first goal is to build the incomplete MIG. Consequently, we use the incomplete MIG representing the old FM as the base for our incremental construction. As we consider the difference of the CNFs as deleting and adding clauses, we only need to inspect the impact of these two types of changes. First, we calculate the impact of these clauses on the core and dead status of features. Second, we check the impact on the redundancy of other clauses. Third, we remove and add the clauses to the MIG and add transitive strong edges. We coordinate all of the aforementioned steps in `adaptMIG` (cf. Algorithm 2). As we can see in line 2, we call `calculateCnfDiff` (cf. Algorithm 1) to calculate the CNF difference as described above. Then, we call `updateDeadAndCoreFeatures` in line 4 where we call a SAT solver to check whether a feature is dead or core, as explained in Section 2.2. Afterwards, we start to calculate the impact on redundant clauses.

3.1.1 Computing Redundant Clauses

Clauses that were redundant in the former CNF were not considered for the generation of the MIG. Thus, we do not need to compute the effect of removing the respective clauses from the MIG after evolution. For that reason, we remove them from the set of removed clauses in line 5 of `adaptMig` (cf. Algorithm 2).

However, clauses that are not redundant potentially have an impact on the redundancy of other clauses. The removal of a clause may cause other clauses to lose their redundancy status. Thus, we need to inspect for all clauses that were redundant if they are still redundant after removing clauses. To reduce the number of required SAT calls, we call `handlePreviouslyRedundant` (cf. Algorithm 3) in line 8 only if any clauses were removed. `HandlePreviouslyRedundant` determines whether a redundant clauses was a *logical consequence* of a set of clauses C that includes at least one of the removed clauses but not a logical consequence of a set of clauses without one of the removed clauses. For a clause to be a logical consequence, every satisfying

Algorithm 2 Algorithm to adapt the MIG to the changes

```
 1: function adaptMig(cnfBeforeChanges, cnfAfterChanges, migBeforeChanges)
 2:     $(removed,\ added) \leftarrow$ calculateCnfDiff($cnfBeforeChanges$,
                                   $cnfAfterChanges$)
 3:     $migAfterChanges \leftarrow migBeforeChanges$
 4:     $migAfterChanges \leftarrow$ updateDeadAndCoreFeatures($cnfAfterChanges$)
 5:     $redundantClauses \leftarrow migBeforeChanges$.getRedundantClauses()
 6:     $removed \leftarrow removed$.removeAll($redundantClauses$)
 7:     if !$removed$.isEmpty() then
 8:         $migAfterChanges$.handlePreviouslyRedundant($redundantClauses$)
 9:     for each $clause \in removed$ do
10:         $migAfterChanges$.removeClause($clause$)
11:     $notRedundantClauses \leftarrow cnfAfterChanges$.removeAll($redundantClauses$)
12:     $affectedClauses \leftarrow$ getAllAffectedClauses($added,\ notRedundantClauses$)
13:     $migAfterChanges$.handleNewlyRedundant($affectedClauses$)
14:     for each $clause \in added$ do
15:         if $clause.size() < 3 \vee$ !isRedundant($clause$) then
16:             $migAfterChanges \leftarrow migAfterChanges$.addClause($clause$)
17:         else
18:             $redundantClauses$.add($clause$)
19:     $migAfterChanges \leftarrow$ dfsStrong($migAfterChanges$)
    return $migAfterChanges$
```

assignment of C has to be a satisfying assignment of the clause. To determine if that is the case, we call a SAT solver in line 3 of `handlePreviouslyRedundant` for every clause. If a clause is not redundant anymore, we call `addClause` (cf. Algorithm 7) in line 4 to add the clause to the MIG. Additionally, we remove it from the set of redundant clauses in line 5.

Algorithm 3 Algorithm to handle previously redundant clauses

```
1: function handlePreviouslyRedundant(clauses)
2:     for each $clause \in clauses$ do
3:         if !isRedundant($clause$) then
4:             $mig$.addClause($clause$)
5:             $redundantClauses$.remove($clause$)
```

Analogously to removing clauses, adding clauses may result in other clauses to become redundant. A clause might be a logical consequence of a set of clauses that includes one of the added clauses. In lines 11 to 13 of `adaptMig` (cf. Algorithm 2), we deal with the effect of added clauses to the redundancy of others. We call `getAllAffectedClauses` (cf. Algorithm 5) in line 12 to figure out which clauses might be affected in their redundancy status by the added clauses. To only consider clauses that contain a literal that is also contained in the added clause does not cover every possible impact. Consider the following clauses: $\{A,\ B,\ D\}$ and $\{\neg X,\ A,\ B\}$. Now we add the unit clause $\{X\}$. Since X is core in consequence of the unit clause, the clause $\{\neg X,\ A,\ B\}$ can now be expressed as $\{A,\ B\}$. Thus, either A or B has to be selected and $\{A,\ B,\ D\}$ is redundant.

We differentiate three cases for the incremental computation of redundant clauses in this thesis:

1. ignore the redundant clauses,
2. search for clauses that have overlapping literals with the removed clauses,
3. or recursively find all clauses that share at least one literal as a clause that contains a literal from the removed clause and so on.

We do so, because the third case, which is the most precise one with regards to the already existing algorithm of the MIG calculated from scratch, costs a lot of computation effort. Using this differentiation we aim to investigate the trade-off between accuracy and computational effort. The second case is an approximation where we inspect the impact of the incremental adaption. We introduce two algorithms: `getThroughLiteralsAffectedClauses` (cf. Algorithm 4) for the second case and `getAllAffectedClauses` (cf. Algorithm 5) as the algorithm for the third case. The difference between these two algorithms is the recursive search from lines 7 to 8 in `getAllAffectedClauses`. We investigate the impact further in Chapter 5.

Algorithm 4 Algorithm to get all clauses that are affected by a same literal by the given set of clauses in the given set of clauses

```
1: function GETTHROUGHLITERALSAFFECTEDCLAUSES(startClauses, possiblyAffectedClauses)
2:     for each $startClause \in startClauses$ do
3:         for each $clause \in possiblyAffectedClauses$ do
4:             if $startClause$.containsAny($clause$.getVariables()) then
5:                 $affectedClauses$.add($clause$)
6:                 $possiblyAffectedClauses$.remove($clause$)
       return $affectedClauses$
```

Algorithm 5 Algorithm to get all clauses that are affected by the given set of clauses in the given set of clauses

```
1: function GETALLAFFECTEDCLAUSES(startClauses, possiblyAffectedClauses)
2:     for each $startClause \in startClauses$ do
3:         for each $clause \in possiblyAffectedClauses$ do
4:             if $startClause$.containsAny($clause$.getVariables()) then
5:                 $affectedClauses$.add($clause$)
6:                 $possiblyAffectedClauses$.remove($clause$)
7:     if !$affectedClauses$.isEmpty() then
8:         $affectedClauses$.add(getAllAffectedClauses($affectedClauses$,
                                   $possiblyAffectedClauses$))
       return $affectedClauses$
```

Both algorithm return clauses that may be affected by one of the added clauses but we do not know if they end up being redundant. To determine that, `handleNewlyRedundant` (cf. Algorithm 6) calculates with a SAT solver for all the possibly affected clauses whether they are redundant after the evolution. In that case, the algorithm removes them from the MIG in line 4 and adds them to the set of redundant clauses in line 5. Once the impact of the added and removed clauses is processed, we still have to add and remove them from the MIG.

Algorithm 6 Algorithm to handle newly redundant clauses

```
1: function HANDLENEWLYREDUNDANT(clauses)
2:     for each $clause \in clauses$ do
3:         if $clause.size() > 2 \wedge$ isRedundant($clause$) then
4:             $mig$.removeClause($clause$)
5:             $redundantClauses$.add($clause$)
```

3.1.2 Handling Added Clauses

In line 18 of `adaptMig` (cf. Algorithm 2) we add the clauses that are new in the CNF either to the MIG or, if they are redundant, to the set of redundant clauses. When adding a clause to the MIG, we call `addClause` (cf. Algorithm 7). To compute the incremental changes of a MIG for an added clause in the CNF, we distinguish between three cases for a different number of literals in the clause, as the algorithm shows. The three cases are:

1. clauses with one literal(i.e., a unit clause),
2. clauses with two literals,
3. and clauses with three or more literals.

We differentiate between these cases because the changes in the MIG depend on the size of the clause. In the following, we discuss the details of the difference.

When we have a clause with one literal, the vertex of the literal needs to be set to dead if the literal is negative and set to core if the literal is positive. A unit clause with a negative literal indicates that the belonging feature cannot be selected in any valid configuration and, thus, is a dead feature. A unit clause with a positive literal indicates that the belonging feature has to be selected in every valid configuration. Accordingly, the feature is core and we express that by setting the vertex of the literal to core. Since we calculate all dead and core feature in `adaptMig`, we do not need to inspect the impact of a unit clause at this point. Apart from that, a clause with one literal has no impact on the MIG, because the MIG displays the dependencies between the literals. Hence, no edges are added to the MIG for a unit clause.

The second case deals with clauses with exactly two literals. Clauses with two literals imply that at least one of the literals needs to be selected for a valid configuration. Hence, it leads to a direct implication between these literals. When one of them is deselected for a configuration, the other literal needs to be selected. For example, changing the or group for the children of Identification to an alternative group in Figure 2.1 would result in the new clause $\{\neg \textit{Online}, \neg \textit{At a polling station}\}$. The clause can be transformed into the logical expression $(\neg \textit{Online} \vee \neg \textit{At a polling station}) \equiv (\textit{Online} \Rightarrow \neg \textit{At a polling station}) \equiv (\textit{At a polling station} \Rightarrow \neg \textit{Online})$. In this example, we would add a strong edge from Online to ¬ At a polling station and from At a polling station to ¬ Online. Thus, two strong edges are added to the MIG. The process is displayed in lines 4 to 10 of `addClause`.

The third case addresses clauses with three or more literals. We deal with that case in lines 12 to 16 of `addClause`. Those clauses express, that at least one of the contained literals has to be selected for a valid configuration. Consider our example from Figure 3.2 which leads to the addition of the clause $\{\neg \textit{By post}, \textit{Duckburg}, \textit{Mouseton}\}$. The clause results in the logical expression $(\neg \textit{By post} \vee \textit{Duckburg} \vee \textit{Mouseton}) \equiv (\textit{By post} \Rightarrow \textit{Duckburg} \vee \textit{Mouseton}) \equiv (\neg \textit{Duckburg} \Rightarrow \neg \textit{By post} \vee \textit{Mouseton}) \equiv (\neg \textit{Mouseton} \Rightarrow \neg \textit{By post} \vee \textit{Duckburg})$. More precisely, deselecting one of the literals implies that at least one of the remaining literals needs to be selected. When adding such a clause, we add weak edges to the MIG. For example, for $(\textit{By post} \Rightarrow \textit{Duckburg} \vee \textit{Mouseton})$ we add weak edges from By post to Duckburg and from By post to Mouseton. The number of weak edges depends on the number of literals in the clause. Let $|L|$ be the number of literals contained in the clause. Then, we have $|L| \cdot (|L| - 1)$ new weak edges in the MIG, what would be six new weak edges for the example. Analogously to the addition of a clause works the removal of a clause.

Algorithm 7 Algorithm for adding a clause to a MIG

```
 1: function ADDCLAUSE(clause, mig)
 2:     $literals \leftarrow clause$.getAllLiterals()
 3:     if $clause$.size() $== 1$ then return
 4:     else if $clause$.size() $== 2$ then
 5:         $firstLiteral \leftarrow literals$.getFirstLiteral()
 6:         $secondLiteral \leftarrow literals$.getSecondLiteral()
 7:         $literalVertex1 \leftarrow mig$.getVertex($firstLiteral$)
 8:         $literalVertex2 \leftarrow mig$.getVertex($secondLiteral$)
 9:         $mig \leftarrow mig$.addStrongEdge($-literalVertex1, literalVertex2$)
10:         $mig \leftarrow mig$.addStrongEdge($-literalVertex2, literalVertex1$)
11:     else
12:         for each $premise \in literals$ do
13:             for each $literal \in literals \setminus \{premise\}$ do
14:                 $premiseVertex \leftarrow mig$.getVertex($premise$)
15:                 $literalVertex \leftarrow mig$.getVertex($literal$)
16:                 $mig \leftarrow mig$.addWeakEdge($-premiseVertex, literalVertex$)
        return $mig$
```

3.1.3 Removal of Clauses in the CNF

All clauses that are contained in the set of removed clauses in `adaptMig` (cf. Algorithm 2) get removed from the MIG in lines 9 and 10 of the algorithm. To this end, we call `removeClause` (cf. Algorithm 8). Similar to the addition of a clause explained in Section 3.1.2, we also differentiate between the cases of one, two, and more than two literals in the clause.

Since clauses with one literal have no effect on the MIG and were not displayed in the MIG in the first place, we do not need to remove them.

After removing a clause with two literals, the direct implication between them does not exist anymore. Thus, the two corresponding strong edges are removed from the MIG in lines 4 to 10. For example, removing the clause $\{\neg$*In person*, $\neg$*By post*$\}$ leads to the removal of the strong edges from In person to ¬By post and from By post to ¬In person.

When removing a clause with three or more literals, the corresponding weak edges are removed from the MIG in lines 11 to 16. For instance, removing the constraint By post ⇒ Duckburg ∨ Mouseton results in removing the clause $\{\neg$*By post, Duckburg, Mouseton*$\}$. To this end, weak edges are removed from By post to Duckburg and to Mouseton, from ¬Duckburg to ¬By post and to Mouseton, and from ¬Mouseton to ¬By post and to Duckburg.

Algorithm 8 Algorithm for removing a CNF Clause from a MIG

```
 1: function REMOVECLAUSE(clause, mig)
 2:     $literals \leftarrow clause$.getAllLiterals()
 3:     if $clause$.size() $== 1$ then return
 4:     else if $clause$.size() $== 2$ then
 5:         $firstLiteral \leftarrow literals$.getFirstLiteral()
 6:         $secondLiteral \leftarrow literals$.getSecondLiteral()
 7:         $literalVertex1 \leftarrow mig$.getVertex($firstLiteral$)
 8:         $literalVertex2 \leftarrow mig$.getVertex($secondLiteral$)
 9:         $mig \leftarrow mig$.removeStrongEdge($-literalVertex1, literalVertex2$)
10:         $mig \leftarrow mig$.removeStrongEdge($-literalVertex2, literalVertex1$)
11:     else
12:         for each $premise \in literals$ do
13:             for each $literal \in literals \setminus \{premise\}$ do
14:                 $premiseVertex \leftarrow mig$.getVertex($premise$)
15:                 $literalVertex \leftarrow mig$.getVertex($literal$)
16:                 $mig \leftarrow mig$.removeWeakEdge($-premiseVertex, literalVertex$)
        return $mig$
```

Now, we reached the end of Algorithm 2 and the only step left is `dfsStrong`. Here, we find all transitive strong edges in the MIG with a depth first search that we adapted from the MIG calculated from scratch. Then, we went through all steps necessary for the incremental construction of an incomplete MIG.

3.2 Complete Modal Implication Graphs

Extending an incomplete MIG to a complete MIG means that the graph includes all possible strong edges. Building a complete graph includes two additional steps. First, we have to find transitive weak edges. Second, we add implicit strong edges.

An implicit strong edge is a weak edge that can be transformed into a strong edge due to other constraints. Since all weak edges could end up being implicit strong edges, we have to consider the transitive weak edges as well. Therefore, we store the transitive weak edges but do not add them to the MIG.

For a better understanding, we consider the evolution step in the feature diagram from Figure 3.2 where we added the constraint By post ⇒ Duckburg ∨ Mouseton. For better readability, Figure 3.3 shows the MIG only with the additional edges that we need for this explanation. Every other new edge is ignored in this graphical representation. Due to the new constraint, we have to add a weak edge from ¬Duckburg to ¬By post. As we can see by the blue arrows, we have a connection from Springfield to In person over ¬Duckburg and ¬By post that includes one weak edge. Thus, there is a transitive weak edge from Springfield to In person which we have to store.

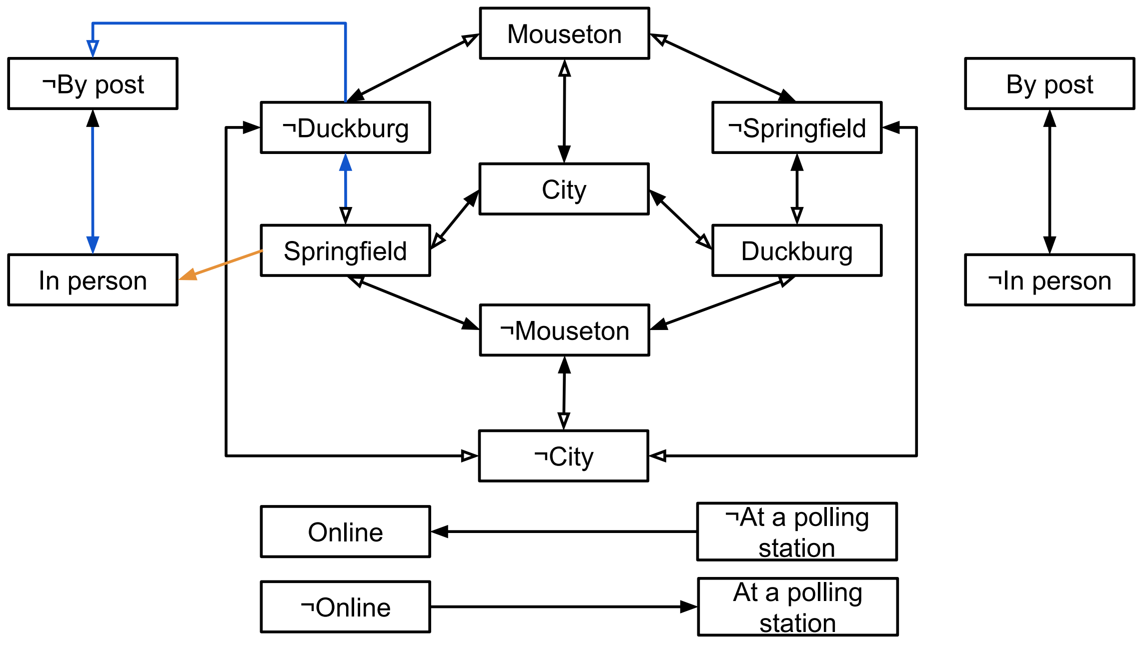


Figure 3.3: Example of a modal implication graph with transitive and implicit edges

The transitive weak edge from Springfield to In person is an implicit strong edge (visualized by the orange edge in Figure 3.3). This is due to the following constraints:

1. Exactly one of By post or In person has to be selected (because of the alternative group which leads to the clauses {*In person, By post*} and {¬*In person*, ¬*By post*}).
2. Only one of Springfield, Duckburg, or Mouseton can be selected (because of the alternative group which leads to the clauses {¬*Duckburg*, ¬*Mouseton*}, {¬*Springfield*, ¬*Mouseton*}, and {¬*Duckburg*, ¬*Springfield*}).
3. When selecting By post, either Duckburg or Mouseton has to be selected (because of the cross-tree constraint which leads to the clause {¬*By post, Duckburg, Mouseton*}).

When selecting Springfield and By post, we have to select Duckburg or Mouseton as consequence of 3. Because of 2, that is not allowed since we already selected Springfield and, thus, By post can not be selected when selecting Springfield. Due to 1, that leads to the selection of In person.

Now we consider the aforementioned transitive weak edge in a virtual clause $\{$*In person,* $\neg$*Springfield*$\}$. We explain, how to figure out whether it is an implicit strong edge on an algorithmical basis. We add both literals contained in the virtual clause negated as a unit clause to the CNF (i.e., we add $\{\neg$*In person*$\}$ and $\{$*Springfield*$\}$). Afterwards, we use a a SAT solver to check whether the CNF is satisfiable with the two unit clauses (i.e., we check whether we can select Springfield when deselecting In person). If the SAT solver returns *false*, it means that at least one of the literals of the virtual clause has to be selected for a valid assignment. Thus, we can add a strong edge for the clause.

For example, the new weak edge between ¬Duckburg to ¬By post also leads to a transitive weak edge between Mouseton and In person. Anyway, this is not an implicit strong edge and also left out in Figure 3.3.

Given a set of clauses that express weak edges that we inspect for implicit strong edges, we call a SAT solver for each pair of literals contained in a clause. To this end, we introduce `findImplicitStrongEdges` (cf. Algorithm 9) which gets the set of clauses as input.

Algorithm 9 Algorithm to check for implicit strong edges in weak edges

```
 1: function FINDIMPLICITSTRONGEDGES(clauses)
 2:     temporaryCNF ← cnfAfterChanges
 3:     for each clause ∈ clauses do
 4:         for each literal ∈ clause do
 5:             temporaryCNF.addClause(-literal)
 6:             for each otherLiteral ∈ clause \ {literal} do
 7:                 temporaryCNF.addClause(-otherLiteral)
 8:                 if !hasSolution(cnfAfterChanges) then
 9:                     migAfterChanges.addClause(literal, otherLiteral)
10:                     migAfterChanges.addImplicitStrongEdge(literal,
                                    otherLiteral)
11:                 temporaryCNF.removeClause(-otherLiteral)
12:             temporaryCNF.removeClause(-literal)
13:             clause.remove(literal)
```

For each pair of literals contained in a clause, we add both literals negated to the CNF in lines 5 and 7. Then, we call a SAT solver to check if the CNF is still satisfiable in line 8. If the SAT call returns false, we add the clause to the MIG in line 9. Afterwards, we remove the unit clauses from the CNF in lines 11 and 12.

Finding implicit strong edges is the most time consuming step when constructing a MIG from scratch [KTS+18]. When constructing the complete MIG incrementally, we calculate the new transitive weak edges from scratch. Afterwards, we have to check if the weak edges that were implicit strong edges in the former MIG are still implicit weak edges. Having a wrong implicit strong edge in the MIG would force the user to (de-)select features in the configuration process that do not need to be (de-)selected. Therefore, missing to delete a former implicit strong edge that is not an implicit strong edge anymore leads to an incorrect MIG. To this end, we call a SAT solver for each implicit strong edge from the former MIG with both of the literals contained in the clause as a negated unit clause. If the SAT solver returns *true* (i.e., the CNF is satisfiable with both literals deselected), we remove the implicit strong edge with `removeClause` (cf. Algorithm 8).

To find new implicit strong edges, we distinguish between two ways where we evaluate the trade-off between accuracy and computational effort in Chapter 5:

1. investigate all new weak edges (including the new transitive weak edges),
2. additionally investigate the weak edges of the clauses affected by the added and removed clauses,

For the first case, we call `findImplicitStrongEdges` (cf. Algorithm 9) with the set of clauses C_N of all new weak edges. For the second case, we call `getThroughLiteralsAffectedClauses` (cf. Algorithm 4) to get a set of clauses C_A. C_A contains all clauses, including new transitive weak edges, that are affected through a shared literal by the added clauses C_N. In this case, because the number of new transitive weak edges is very high we leave them out of C_N and only investigate those that have a shared literal with the added clauses. We remove all clauses with two literals from C_A since they already represent strong edges. Then, we add the remaining clauses in C_A to C_N. Afterwards, we call `findImplicitStrongEdges` with C_N as parameter.

Now, we described all steps necessary to extend the incomplete MIG to a complete MIG. With the introduced concepts and algorithms, it is now possible to incrementally calculate MIGs after FM evolution.

4. Implementation

In this chapter, we explain the implementation of the concepts introduced in Chapter 3. To this end, we give an overview of the open-source framework FeatureIDE. Therein, we integrated the incremental construction of a MIG since, to the best of our knowledge, FeatureIDE is the only feature modeling toolsuite that provides a MIG calculation. Afterwards, we point out the adaptions we made in the MIG that is calculated from scratch. These adaptions are necessary for the incremental construction. Finally, we explain how the incremental constructed MIG is implemented.

4.1 FeatureIDE

FeatureIDE is a tool for feature-driven development based on the open-source IDE Eclipse [KTS+09, MTS+17, TKB+14]. FeatureIDE, provides a graphical interface to edit feature models as depicted by Figure 2.1 and Figure 3.2, and various analyses (e.g., detecting dead features). The tool also supports the process of defining configurations. Hereby, mandatory (de-)selections with regards to the current combination of features are performed automatically. As illegal selections are disabled, it is not possible to construct an invalid configuration. Figure 4.1 shows an example of a valid configuration based on the feature model of our running example (cf. Figure 2.1). As we can see, By post is grayed out since In person was selected by the user and only one of them can be selected at the same time. To compute the mandatory selections, FeatureIDE uses a combination of a MIG and a SAT solver.

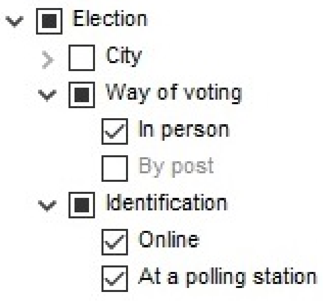


Figure 4.1: Example of a configuration

4.2 Integration of Incremental MIG

Figure 4.2 shows the class diagram of the MIG implemented in FeatureIDE. It contains a list *adjacencyList* that stores the vertices and a list *complexClauses* that stores all clauses of weak edges (i.e., clauses with three or more literals). A vertex stores the *complexClauses* and the *strongEdges* for each variable and whether it is core or dead. As introduced in Chapter 3, the incremental construction adapts the edges and clauses of the MIG, the redundant clauses, the transitive weak edges, the transitive strong edges and the implicit strong edges. This means, that the implementation of the former MIG is not sufficient for the incremental construction since it does not contain the required information about the constructed MIG (i.e., the redundant clauses, the transitive edges, and the implicit strong edges that the incrementally calculated MIG needs to adapt).

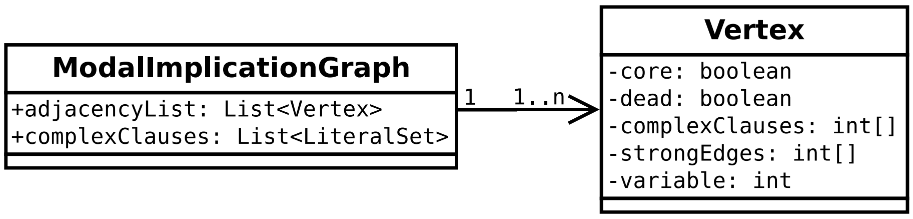


Figure 4.2: Class diagram of the MIG implemented in FeatureIDE

4.2.1 Necessary Adaptions

In this subsection, we point out the adaptions we made in the class ModalImplicationGraph that are necessary to be able to incrementally construct a MIG. Figure 4.3 shows the class diagram of the MIG after our adaptions for the incremental calculations. We save *transitiveStrongEdges*, *transitiveWeakEdges*, *implicitStrongEdges* and *redundantClauses* of the MIG. Additionally, we added multiple methods to the class.

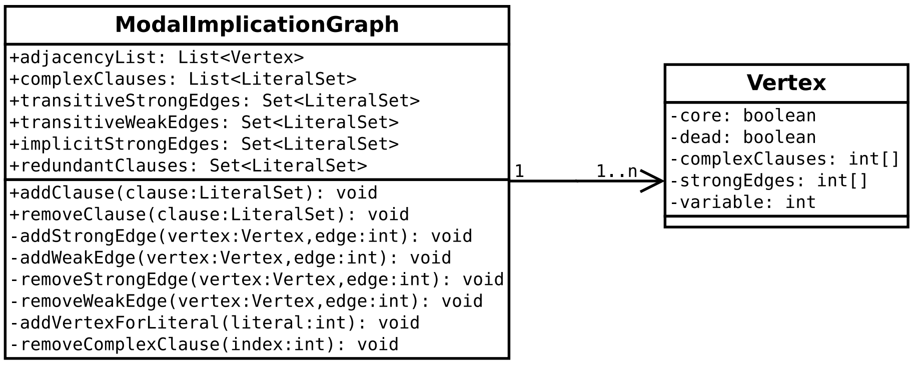

ModalImplicationGraph
+adjacencyList: List<Vertex>
+complexClauses: List<LiteralSet>
+transitiveStrongEdges: Set<LiteralSet>
+transitiveWeakEdges: Set<LiteralSet>
+implicitStrongEdges: Set<LiteralSet>
+redundantClauses: Set<LiteralSet>
+addClause(clause:LiteralSet): void
+removeClause(clause:LiteralSet): void
-addStrongEdge(vertex:Vertex,edge:int): void
-addWeakEdge(vertex:Vertex,edge:int): void
-removeStrongEdge(vertex:Vertex,edge:int): void
-removeWeakEdge(vertex:Vertex,edge:int): void
-addVertexForLiteral(literal:int): void
-removeComplexClause(index:int): void
1
1..n
Vertex
-core: boolean
-dead: boolean
-complexClauses: int[]
-strongEdges: int[]
-variable: int

Figure 4.3: Class diagram of the new MIG

First, we call `addClause` (cf. Algorithm 7) when adding a clause to the MIG. This method checks whether the literals in the clause already have a vertex. If they are new to the MIG, a vertex is added with `addVertexForLiteral`. Afterwards, depending on the size of the clause `addClause` returns or calls either `addStrongEdge` or `addWeakEdge`.

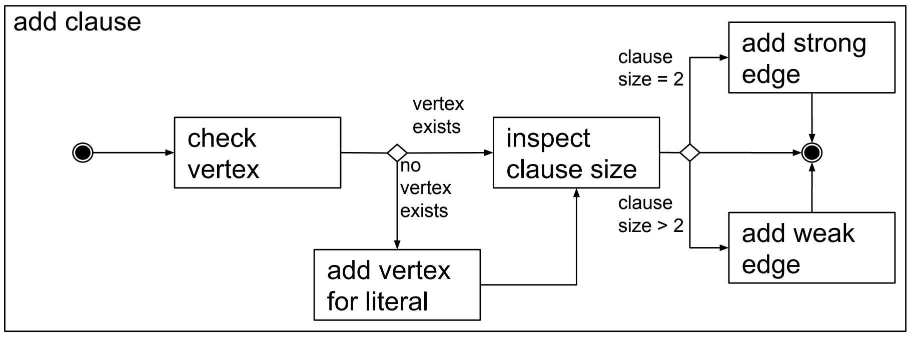


Figure 4.4: Steps for adding a clause

Second, we call `removeClause` (cf. Algorithm 8) when removing a clause from the MIG. Again, we distinguish between the size of the clause and return or call either `removeStrongEdge` or `removeWeakEdge`. When removing a weak edge, we have to consider that the associated complex clause is not only saved in the vertices but also in the MIG itself. Hence, `removeClause` also calls `removeComplexClause` when removing a weak edge to (1) remove the complex clause from the MIG and (2) make sure that the indices of the complex clauses are still saved correctly for the contained vertices after the removal of one of the complex clauses.

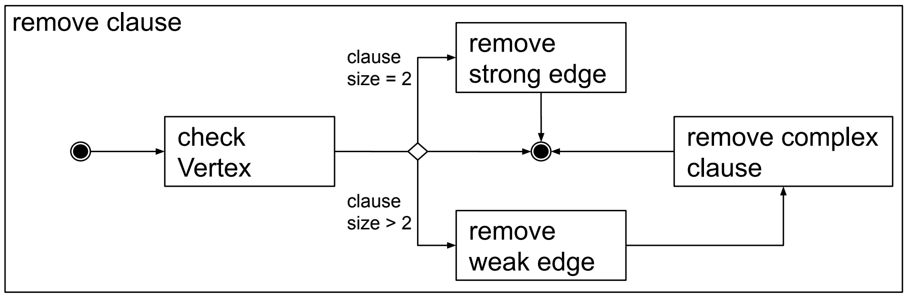


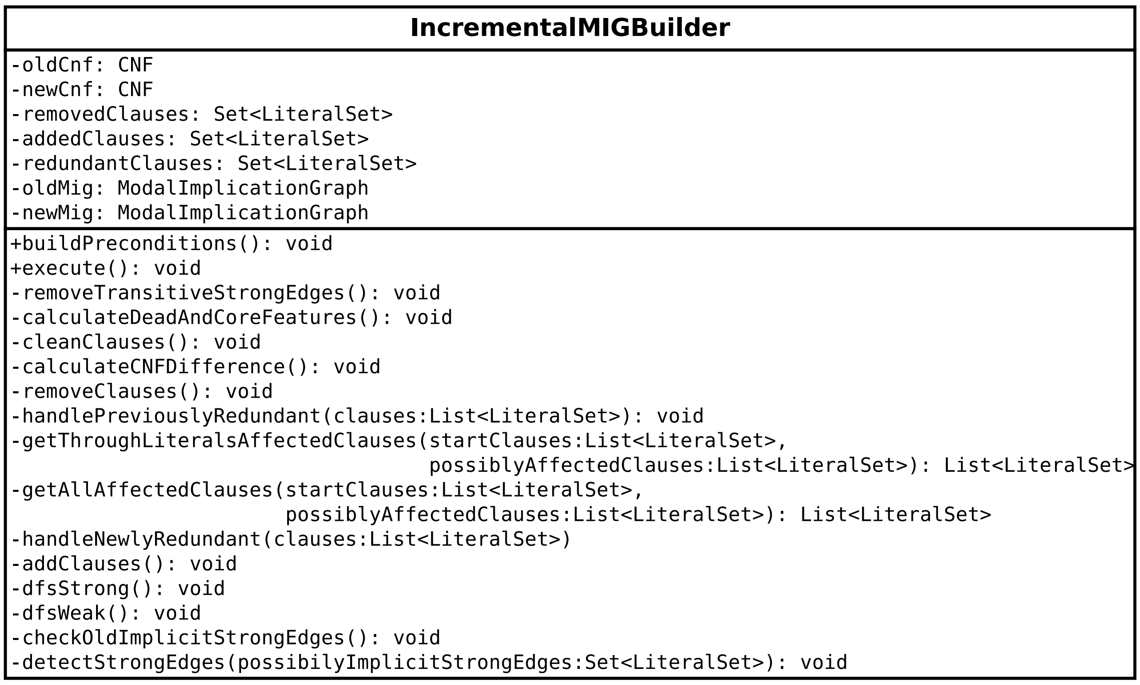
Figure 4.5: Steps for removing a clause

4.2.2 Incremental MIG

In this subsection, we present the class IncrementalMIGBuilder that contains the implementation for incrementally constructing incomplete and complete MIGs. Figure 4.6 shows the class diagram for this specific class.

The first method of this class is `buildPreconditions`. Here, we determine the *oldCnf* of the *oldMig* where we use the CNF that resulted from sorting out unnecessary clauses. We also determine the *newCnf* of the FM after the evolution step. We also copy the values of the old MIG into the variable *newMig*, which we adapt in the following steps.

The second method we call is `execute` which coordinates all steps and calls all of the methods in the following order.



```
IncrementalMIGBuilder
-oldCnf: CNF
-newCnf: CNF
-removedClauses: Set<LiteralSet>
-addedClauses: Set<LiteralSet>
-redundantClauses: Set<LiteralSet>
-oldMig: ModalImplicationGraph
-newMig: ModalImplicationGraph
+buildPreconditions(): void
+execute(): void
-removeTransitiveStrongEdges(): void
-calculateDeadAndCoreFeatures(): void
-cleanClauses(): void
-calculateCNFDifference(): void
-removeClauses(): void
-handlePreviouslyRedundant(clauses:List<LiteralSet>): void
-getThroughLiteralsAffectedClauses(startClauses:List<LiteralSet>,
                                   possiblyAffectedClauses:List<LiteralSet>): List<LiteralSet>
-getAllAffectedClauses(startClauses:List<LiteralSet>,
                       possiblyAffectedClauses:List<LiteralSet>): List<LiteralSet>
-handleNewlyRedundant(clauses:List<LiteralSet>)
-addClauses(): void
-dfsStrong(): void
-dfsWeak(): void
-checkOldImplicitStrongEdges(): void
-detectStrongEdges(possibilyImplicitStrongEdges:Set<LiteralSet>): void
```

Figure 4.6: Class diagram of the IncrementalMIGBuilder

1. `RemoveTransitiveStrongEdges`. This method removes all deprecated strong edges from the new MIG. The methods operates on the old transitive strong edges from *transitiveStrongEdges* stored in ModalImplicationGraph. `DfsStrong` (i.e., the depth first search for finding transitive strong edges) finds all possible transitive strong edges and, thus, it is not necessary to distinguish between deprecated and valid strong edges in this step. We have to remove the strong edges, because wrong strong edges in the MIG lead to an incorrect MIG.

2. `CalculateDeadAndCoreFeatures`. This method is adopted from MIGBuilder, the class that coordinates the steps for calculating a MIG from scratch. It calculates the core and dead features in *newCnf* and changes the respective vertices.

3. `CleanClauses`. This method is adopted from MIGBuilder as well and sorts out unnecessary clauses in *newCnf* (e.g., clauses that contain a core feature). This reduces the MIG to a size where it only contains the necessary clauses. Additionally, it simplifies the following steps (e.g., `calculateCNFDifference` is faster when using smaller CNFs).

4. `CalculateCNFDifference`. This method calculates the difference between *oldCnf* and *newCnf* (cf. Algorithm 1). It stores the determined clauses in *removedClauses* and *addedClauses* which we use later on.

5. `RemoveClauses`. This method removes the edges of the clauses that are contained in *removedClauses* (cf. Algorithm 8). We also have to consider clauses that were redundant in the former MIG and are in the set of removed clauses.

To this end, we remove all clauses that are contained in *redundantClauses* of the *oldMig* from *removedClauses*. Also, we remove all clauses that are contained in *removedClauses* from the *redundantClauses* of *newMig*.

6. `HandlePreviouslyRedundant`. This method checks for each *redundantClause* of *oldMig* if it still is a redundant clause (cf. Algorithm 3). If it is not, the method calls `addClause` for adding the specific clause to the *newMig* and removes the clause from the set of redundantClauses. This step is necessary to prevent clauses to be declared redundant even when their redundancy changed due to the FM evolution.

7. `GetThroughLiteralsAffectedClauses`. This method gets two sets of clauses as parameters, i.e., *startClauses* and *possiblyAffectedClauses* (cf. Algorithm 4). It returns a list of all clauses in *possiblyAffectedClauses* that contain a literal that is also contained in a clause in *startClauses*. With this method we find the clauses that are most likely to be affected by the FM evolution regarding (1) their redundancy and (2) to contain implicit strong edges due to added clauses.

8. `GetAllAffectedClauses`. This method gets the same parameters as `get-ThroughLiteralsAffected` (cf. Algorithm 5). But instead of returning clauses that share a literal with an added clause, it returns a list of all clauses in *possiblyAffectedClauses* that have any connection over literals with a clause that is contained in *startClauses*. With this method we find all clauses that are affected by the FM evolution. `getAllAffectedClauses` has a higher degree of accuracy than `getThroughLiteralsAffectedClauses` regarding the similarity to the from scratch calculated MIG.

9. `HandleNewlyRedundant`. This method gets the clauses that were returned by either `getThroughLiteralsAffectedClauses` or `getAllAffectedClauses` (depending on the desired degree of accuracy) as input (cf. Algorithm 6). Then, it checks which of them are now redundant after the FM evolution step.

10. `AddClauses`. This method adds all clauses from *addedClauses* that are not redundant to the MIG by calling `addClause` for each clause. Otherwise, it adds them to *redundantClauses* of *newMig*.

11. `DfsStrong`. This method is adapted from MIGBuilder to fit into our implementation (i.e., we use the *adjancencyList* of the MIG instead of an adjacency matrix that the MIGBuilder uses). With a depth first search, this method searches the MIG to find literals that have a connection over at least two strong edges. It adds a strong edge between respective literals.

By executing the previously introduced methods, we incrementally calculate an incomplete MIG. As we can see, the class diagram in Figure 4.6 contains three additional methods. These methods are ignored when building an incomplete graph and are only executed when calculating a complete graph.

Extensions for complete MIG

To make an incomplete MIG complete, we need to calculate implicit strong edges which is done using the following methods:

1. `DfsWeak`. This method is adapted from MIGBuilder, but we again use the *adjacencyList* instead of a matrix. This method uses a depth first search to find literals that have a connection over at least two edges containing at least one weak edge. We store these edges for the detection of implicit strong edges.

2. `CheckPreviouslyImplicitStrongEdges`. This method checks for all *implicitStrongEdges* of the *oldMig* whether they are still implicit strong edges after the FM evolution.

3. `DetectStrongEdges`. This method finds new implicit strong edges. As input it gets a set of possibly implicit strong edges. Depending on the degree of accuracy, the set contains either the added clauses plus the new transitive weak edges, or the added clauses plus the clauses affected by the added clauses and the new transitive weak edges.

In this section, we explained our implementation to incrementally construct MIGs as an extension of the already existing, from scratch calculated MIG in FeatureIDE. To this end, we provide implementations for the concepts and algorithms Chapter 3 that result in a logically correct MIG. The empirical evaluation presented in Chapter 5 is based on this implementation.

5. Evaluation

In this chapter, we examine the benefits of the incremental construction introduced in Chapter 3 based on the implementation described in Chapter 4. To this end, we introduce three research questions and answer the questions with an empirical evaluation of our method. For our evaluation, we use feature models of different sizes and different numbers of evolution steps to investigate how much the efficiency of the incremental construction depends on (1) the size of the feature model and (2) the number of changes in the FM and, thus, in the CNF. For each research question, we compare the incremental constructed MIG with the MIG constructed from scratch. After presenting the results of the evaluation in Section 5.2, we discuss them in Section 5.3.

5.1 Experiment Design

In the following, we describe the software and hardware we use in our evaluation:

- Operating system: Windows 10
- Eclipse version: Eclipse Modeling Tools version 2020-03 (4.15.0)
- FeatureIDE version: FeatureIDE version 3.7
- RAM: 16GB
- Architecture: 64-bit
- CPU: AMD Ryzen 7 3700U, 2300MHz, 4 cores

In the following, we present the FMs and their respective versions that we use for our evaluation:

1. *Automotive02*. These are by far the largest FMs and the most added/removed clauses in the CNF that we evaluate in this work. We have three different FM evolution steps that we evaluate:

FM evolution	Number of clauses	Number of added clauses	Number of removed clauses
V1 to V2	237,698 to 342,923	241,635	136,410
V2 to V3	342,923 to 347,541	5,366	748
V3 to V4	347,541 to 350,180	2,913	274

Table 5.1: All FM evolution steps for the Automotive02 FM versions.

2. *FinancialServices01.* The evaluation consists of nine FM versions that are collected from monthly commits, whereas the first and the second FM version have four months in between.

FM evolution	Number of clauses	Number of added clauses	Number of removed clauses
2017-05-22 to 2017-09-28	4,992 to 6,544	2,271	719
2017-09-28 to 2017-10-20	6,544 to 6,778	255	21
2017-10-20 to 2017-11-20	6,778 to 6,866	173	85
2017-11-20 to 2017-12-22	6,866 to 6,860	107	113
2017-12-22 to 2018-01-23	6,860 to 6,682	2,599	2,777
2018-01-23 to 2018-02-20	6,682 to 6,792	320	210
2018-02-20 to 2018-03-26	6,792 to 7,112	667	347
2018-03-26 to 2018-04-23	7,112 to 7,134	181	159
2018-04-23 to 2018-05-09	7,134 to 7,238	300	196

Table 5.2: All FM evolution steps for the FinancialServices01 FM versions.

3. *BusyBox.* Here we have 3,712 FM versions that were collected by daily commits from 20.05.2007 until 09.05.2010 with around 1,000 clauses in the CNF.

To cover all algorithms we introduced in Chapter 3, we distinguish between nine different calculations that we evaluate:

1. Incomplete MIG without the calculation of redundant clauses
2. Incomplete MIG with the calculation of redundant clauses, where we only inspect through literals affected clauses in the incremental construction

3. Incomplete MIG with the calculation of redundant clauses, where we inspect all affected clauses
4. Complete MIG without the calculation of redundant clauses and with only the new weak and the new transitive weak edges as possibly implicit strong edges
5. Complete MIG without the calculation of redundant clauses and with additionally affected clauses as possibly implicit strong edges
6. Complete MIG with the calculation of redundant clauses as in 2. and with only the new weak and the new transitive weak edges as possibly implicit strong edges
7. Complete MIG with the calculation of redundant clauses as in 2. and with additionally affected clauses as possibly implicit strong edges
8. Complete MIG with the calculation of redundant clauses as in 3. and with only the new weak and the new transitive weak edges as possibly implicit strong edges
9. Complete MIG with the calculation of redundant clauses as in 3. and with additionally affected clauses as possibly implicit strong edges

In the following, we introduce the research questions of this thesis:

RQ1: *How much computation time can we save with the incremental construction of a MIG?*

In the first research question, we investigate the computation time of the incremental construction. In this research question, we inspect the advantages of the incremental calculation compared with the calculation from scratch regarding the computation time. To this end, we compare (1) the times that both calculations need to construct a MIG and (2) the times that both calculation need for each step when constructing a MIG.

RQ2: *How many SAT queries can we reduce with the incremental construction?*

In the second research question, we check how many SAT queries are saved with our method. Since the goal of this thesis was to reduce the number of SAT calls that the calculation from scratch needs when constructing a MIG, we investigate whether we achieve that goal with the incremental construction. To this end, we count how often we call a SAT solver when (1) incrementally constructing the MIG and (2) constructing the MIG from scratch. Afterwards, we compare the two results with each other.

RQ3: *How does the size of the incrementally constructed MIG compare to the MIG constructed from scratch?*

In the third and last research question, we inspect the difference between the resulting MIGs. This is necessary to find out the degree of accuracy of the incremental construction. To this end, we investigate the number of redundant clauses and the number of implicit strong edges since these are the only differences that may appear.

5.2 Results

In the following, we show the results of our empirical evaluation for each research question. First, we present the results of our evaluation regarding the incomplete graph. Second, we show the results regarding the complete graph. To this end, we evaluate and compare the necessary calculation time needed for the incremental construction and the calculation from scratch for all BusyBox models, FinancialServices01 models, and Automotive02 models. Additionally, we compare the times that each step (e.g., find redundant clauses, find transitive edges) in the calculation process needs for some examples. The x-axis of all following diagrams lists the FM evolution step for which we construct the MIG and the y-axis shows, depending on the research question, the calculation time required for that construction, the number of SAT calls, the number of redundant clauses, or the number of implicit strong edges. The data for the incremental construction is illustrated with green circles and for the calculation from scratch, it is illustrated with blue triangles.

RQ1

To retrieve the computation time we save, we measure the time between the start and the end of the calculation for the incremental MIG as well as its calculation from scratch. Then, we compare these two times with each other. We also measure the time of the single steps (e.g., calculating the redundant clauses, adding and removing the edges, calculating core and dead features,..) for both methods and compare them. Before each calculation, we warm up the java virtual machine (JVM) by running the MIG calculations on ten FM evolution steps.

Incomplete Graph

In the following, we present the results for incomplete MIGs without redundant clauses. Figure 5.1 shows the required times of the calculation from scratch and the

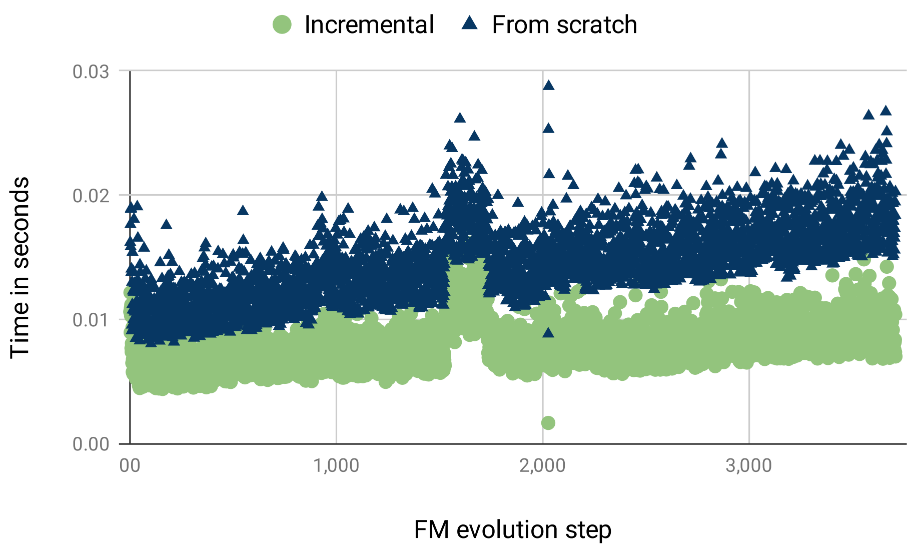


Figure 5.1: Calculation times for all BusyBox evolution steps for an incomplete MIG without redundant clauses.

incremental construction for all BusyBox evolution steps. As the diagram shows, the incremental construction takes less time than the calculation from scratch for each of the 3,711 FM evolution steps. The results show that, except a few outliers, the incremental construction takes about half of the time or less than the calculation from scratch.

Figure 5.2 shows the required times of the calculation from scratch and the incremental construction for all FinancialServices01 evolution steps. For each FM evolution step, the incremental construction takes between 1.5 and 2 seconds and the construction from scratch takes around 0.5 seconds.

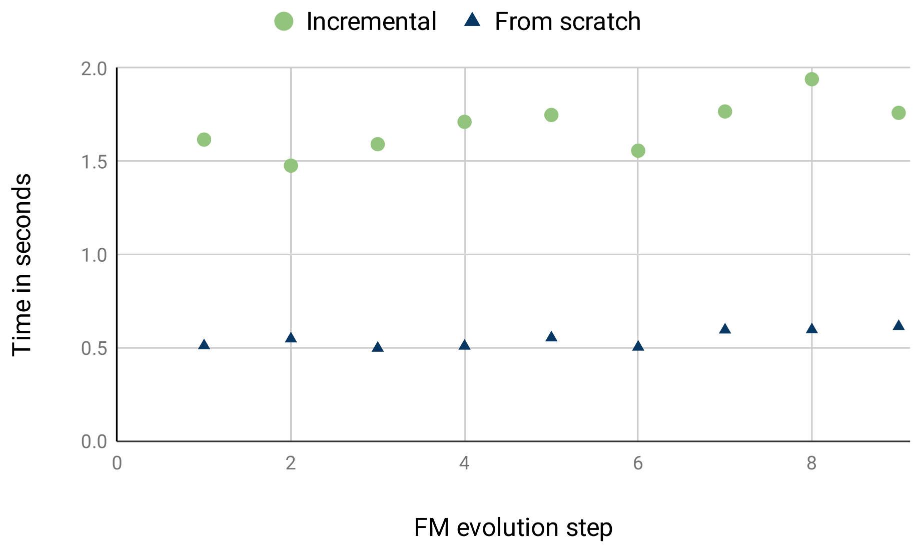


Figure 5.2: Calculation times for all FinancialServices01 evolution steps for an incomplete MIG without redundant clauses.

Figure 5.3 shows the required times of the calculation from scratch and the incremental construction for all Automotive02 evolution steps. Incrementally calculating

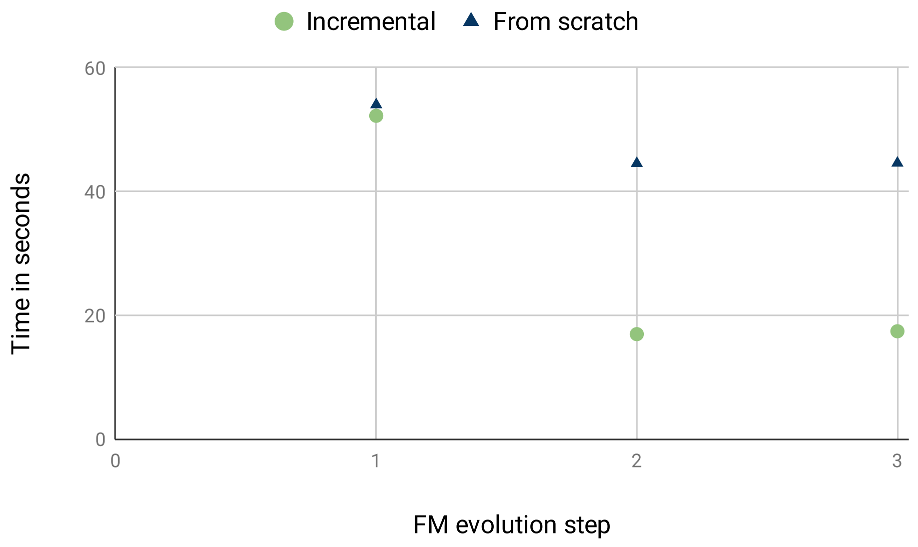


Figure 5.3: Calculation times for all Automotive02 evolution steps for an incomplete MIG without redundant clauses.

Automotive02_V2 with Automotive02_V1 as base takes just a few seconds less than calculating it from scratch. Anyway, the incremental construction of Automotive02_V3 and Automotive 02_V4 takes less than half of the calculation time needed to calculate the MIG from scratch.

To get a better overview where the computation time difference comes from, we take a closer look at the computation times each step takes in Table 5.3. We chose the evolution step from Automotive02_V2 to Automotive02_V3 as representative. The runtime distribution for other evolution steps and models is similar.

Calculation Step	Calculated MIG (sec)	Adapted MIG (sec)	Time Difference Adapted MIG - Calculated MIG(sec)
Remove old transitive strong edges	0	0.09	+0.09
Calculate core and dead feature	9.18	9.08	-0.1
Clean CNF	0.19	0.11	-0.07
Calculate CNF difference	0	0.52	+0.52
Removed clauses	0	0.005	+0.005
Added clauses	0	1.04	+1.04
Create MIG	2.25	0	-2.25
Calculate redundant clauses	0.03	0	-0.03
Find transitive strong edges	32.77	6.0	-26.77
Overall time	44.53	16.93	-27.59

Table 5.3: Calculation times for each step for the incomplete MIG without redundant clauses in the evolution step from Automotive02_V2 to Automotive02_V3.

The table shows that the additional steps for the incremental construction (remove old transitive strong edges, calculate CNF difference, remove clauses, and add clauses) take around a half second less time as calculating the MIG from scratch (1.655 seconds vs. 2.25 seconds). The most time is saved when calculating transitive strong edges in the MIG. In the end, the incrementally calculated MIG is 27.59 seconds faster, whereas 26.77 seconds derive from the calculation of transitive strong edges.

In the following, we present the evaluation of the incomplete graph with the calculation of redundant clauses, where we only include clauses that are affected through literals, as we introduced in Section 3.1.1.

Figure 5.4 shows the calculation times for all BusyBox models. Except a few outliers, the incrementally constructed MIG takes around 0.01 seconds and the required time for the from scratch calculated MIG grows with the FM versions from around 0.02-0.03 seconds to around 0.04 seconds. Comparing to the calculation without redundant clauses, the time needed for the incremental construction is the same but the time needed for the MIG calculated from scratch increases.

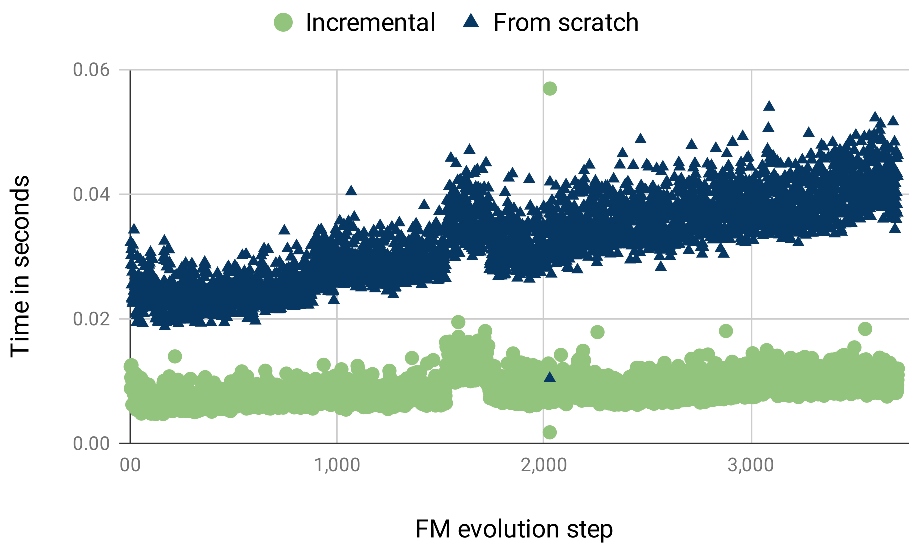


Figure 5.4: Calculation times for all BusyBox evolution steps for an incomplete MIG with redundant clauses.

Figure 5.5 shows the FinancialServices01 calculation times. While the incremental construction without redundant clauses takes more than three times as much as the calculation from scratch, the calculation with redundant clauses takes for the incremental construction more than double of the time that the calculation from scratch takes.

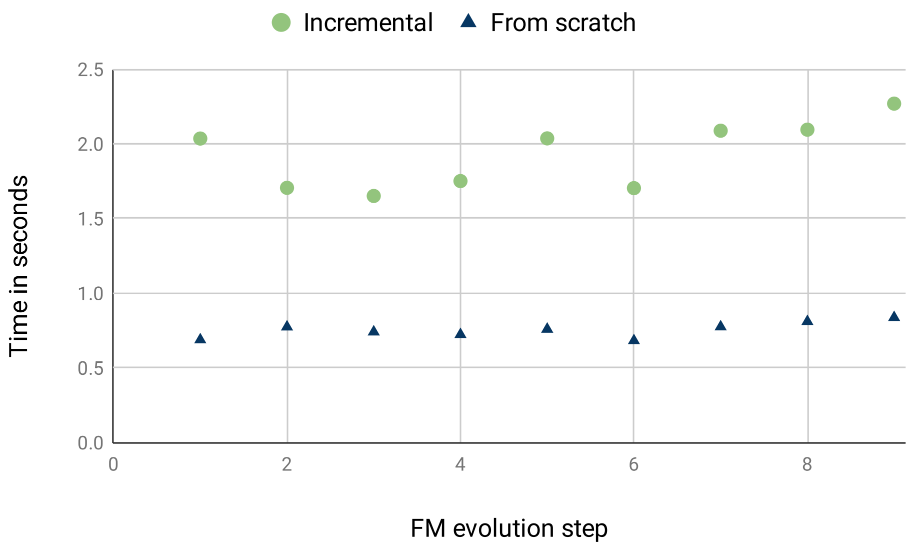


Figure 5.5: Calculation times for all FinancialServices01 evolution steps for an incomplete MIG with redundant clauses.

With Figure 5.6, we present the calculation times for Automotive02, the largest FMs in our evaluation. In the first evolution step, from Automotive02_V1 to Automotive02_V2, the incremental construction does not save a lot of time, but in the second and third evolution step, the incremental construction takes less than half of the time the calculation from scratch requires.

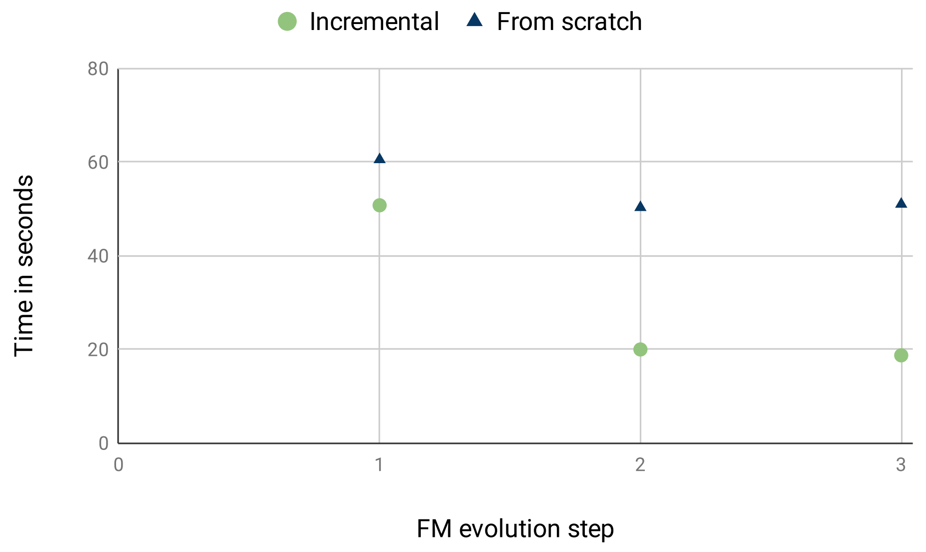


Figure 5.6: Calculation times for all Automotive02 evolution steps for an incomplete MIG without redundant clauses.

Table 5.4 depicts the required time of each step for Automotive02_V2 to Automotive02_V3. Similar to the results presented in Table 5.3, the most time is saved during the calculation of the transitive strong edges. Additionally, the incremental construction improves the calculation of redundant clauses. We have to consider, that some of the redundant clauses are calculated when adding clauses. Anyway, when comparing the time necessary for adding clauses to the calculation without redundant clauses, we can see that $\sim$ 1.7 seconds are required in addition. With regards to these seconds, the calculation of redundant clauses is still three seconds faster when calculating incrementally.

When calculating the MIG including the redundant clauses with all affected clauses as introduced in Section 3.1.1, too much memory space is required. With 12GB of RAM available, BusyBox, FinancialServices01 and Automotive02 throw a `java.lang.OutOfMemoryError: Java heap space` Exception. Thus, we could not evaluate these calculations.

Complete Graph

Second, we show the results of the evaluation regarding the complete graph. We take a look at the calculation of complete MIGs including redundant clauses. We also present one representative example for the calculation without redundant clauses for the BusyBox FMs in Figure 5.7.

We can see the computation times for BusyBox for the complete graph in Figure 5.8. The times are similar to the calculation without redundant clauses. As we can see, the complete MIG takes much more computation time compared to the incomplete MIG. While the calculation from scratch takes maximum a half second for the incomplete MIG, for the complete MIG early versions require a half second and, as

Calculation Step	Calculated MIG (sec)	Adapted MIG (sec)	Time Difference Adapted MIG - Calculated MIG(sec)
Remove old transitive strong edges	0	0.09	+0.09
Calculate core and dead feature	9.43	9.65	+0.22
Clean CNF	0.19	0.11	-0.08
Calculate CNF difference	0	0.49	+0.49
Removed clauses	0	0.007	+0.007
Added clauses	0	2.74	+2.74
Create MIG	1.86	0	-1.86
Calculate redundant clauses	5.11	1.42	-3.69
Find transitive strong edges	32.24	5.8	-26.44
Overall time	49.22	20.37	-28.84

Table 5.4: Calculation times for each step for the incomplete MIG with redundant clauses in the evolution step from Automotive02_V2 to Automotive02_V3.

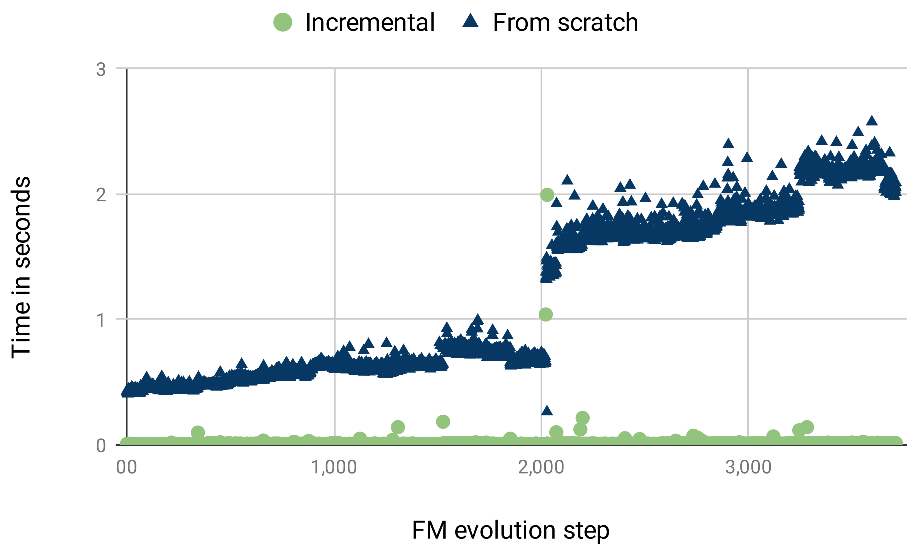


Figure 5.7: Calculation times for all BusyBox evolution steps for a complete MIG without redundant clauses.

the FM versions grow, require up to over 2 seconds. The calculated average is 1.121 seconds. However, incrementally constructing the MIG takes on the average 0.009 seconds and, thus, is ∼125x faster on average than calculating the MIG from scratch.

Figure 5.9 shows the calculations for the FinancialServices01 models. In this diagram, the time is given in minutes on the y-axis. We can see that the incremental construction for the first and the fifth evolution step take a lot of time (between eight and ten minutes). Table 5.2 shows that the first and fifth evolution step include large

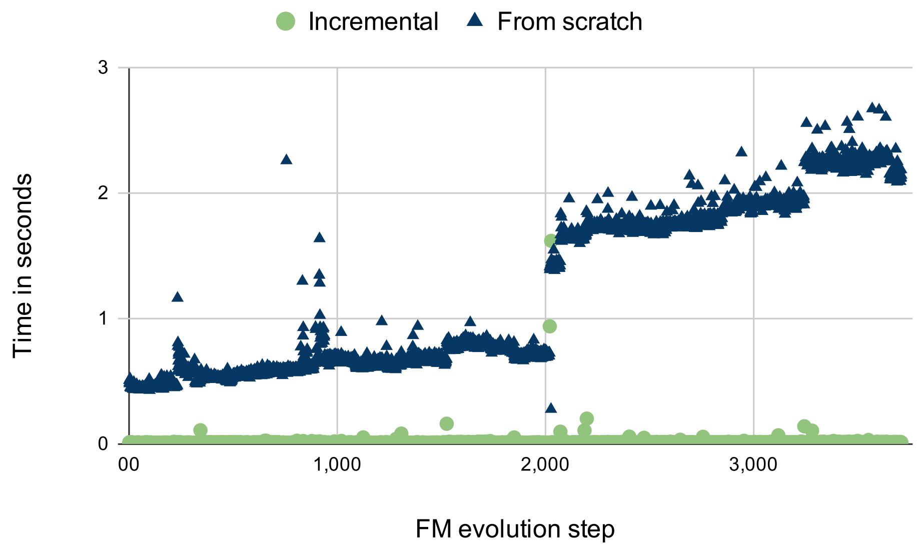


Figure 5.8: Calculation times for all BusyBox evolution steps for a complete MIG.

changes in the FM. However, the other incrementally calculated evolution steps are multiple minutes faster than calculating the MIG from scratch. Overall, the incremental calculation requires ∼15 1/2 minutes and the calculation from scratch ∼38 minutes for the eight evolution steps.

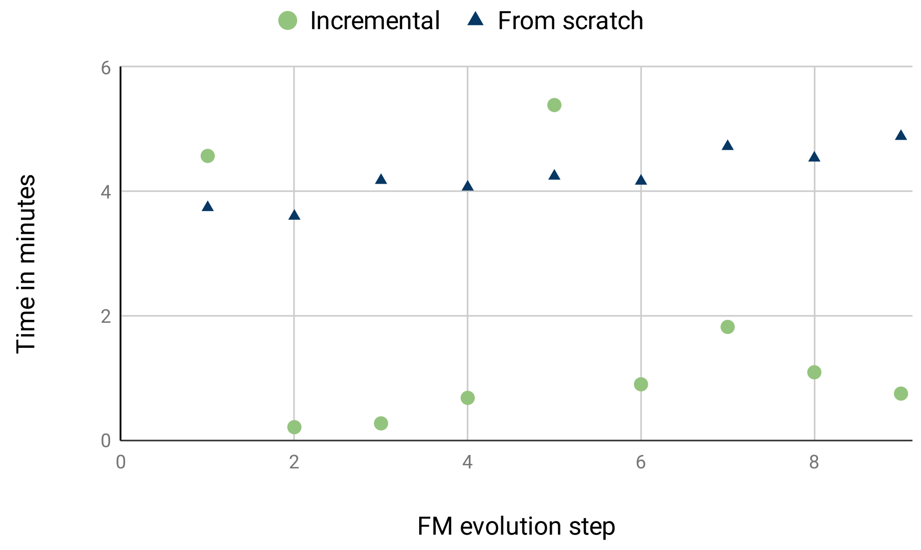


Figure 5.9: Calculation times for all FinancialServices01 evolution steps for a complete MIG.

Figure 5.10 shows the calculation for all Automotive02 versions. Again, the y-axis shows the time in minutes. We can see that in the first step, the incremental calculation takes more than double of the time that the calculation from scratch takes. However, in the second and third evolution step, the incrementally calculated MIG takes much less computation time (i.e., ∼5.8x faster for the second step and ∼11.8x faster for the third step).

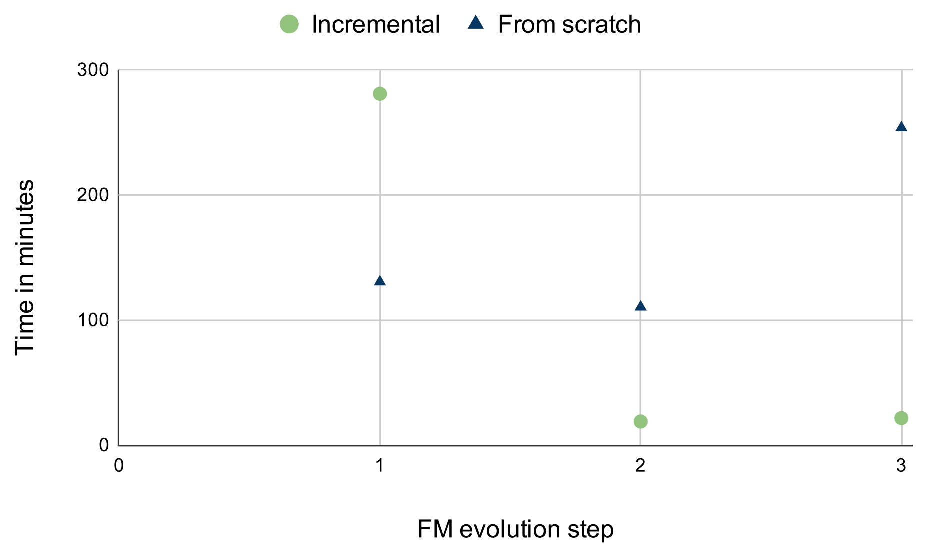


Figure 5.10: Calculation times for all Automotive02 evolution steps for a complete MIG.

Calculation Step	Calculated MIG (sec)	Adapted MIG (sec)	Time Difference Adapted MIG - Calculated MIG(sec)
Remove old transitive strong edges	0	0.08	+0.08
Calculate core and dead feature	9.38	9.33	+0.05
Clean CNF	0.24	0.11	-0.13
Calculate CNF difference	0	0.43	+0.43
Check previously implicit strong edges	0	0.96	+0.96
Removed clauses	0	0.003	+0.003
Added clauses	0	0.83	+0.83
Find transitive strong edges	34.05	5.16	-28.89
Find transitive weak edges	43.56	0.86	-42.7
Find implicit strong edges	14,881.84	1,267.71	-13,614.13
Create MIG	2.07	0	-2.07
Calculate redundant clauses	5.47	0.89	-4.58
Find transitive strong edges	33.77	5.2	-28.57
Overall time	15,010.45	1,291.63	-13,718.82

Table 5.5: Calculation times for each step for the complete MIG in the evolution step from Automotive02_V3 to Automotive02_V4.

Finally, we take a look at the exact times that each calculation step takes for the complete graph in Table 5.5. As the table shows, finding implicit strong edges is the most time consuming step and also the biggest difference between the two calculations. The calculation from scratch takes ca 248 minutes (14,881 seconds) and the incremental construction takes ca 21 minutes (1,267 seconds). This leads to the incremental construction being ca 226 minutes faster, only considering this specific calculation step. Overall, the incremental construction takes 228 minutes less than the calculation from scratch.

Figure 5.11 and Figure 5.12 show the calculation times needed for the complete graph without affected clauses for BusyBox and FinancialServices01. When comparing to

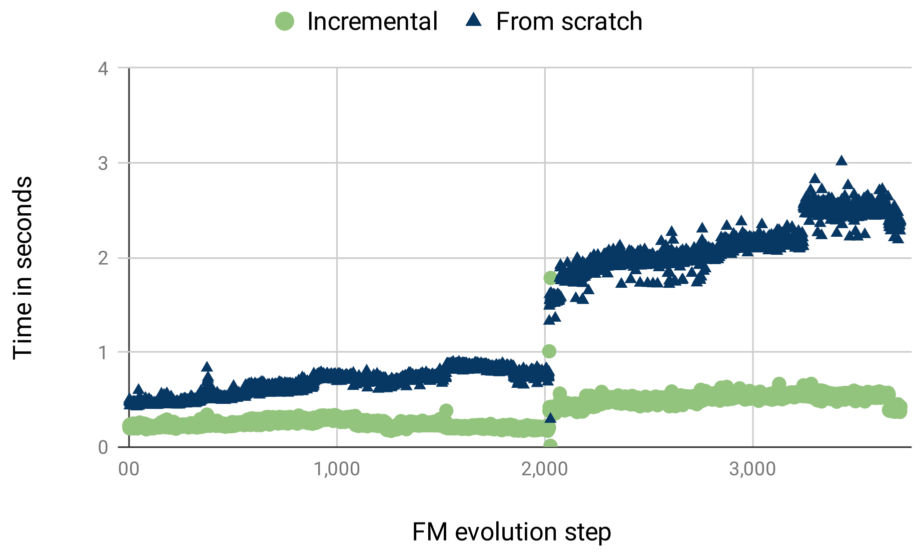


Figure 5.11: Calculation times for all BusyBox evolution steps for a complete MIG without affected clauses.

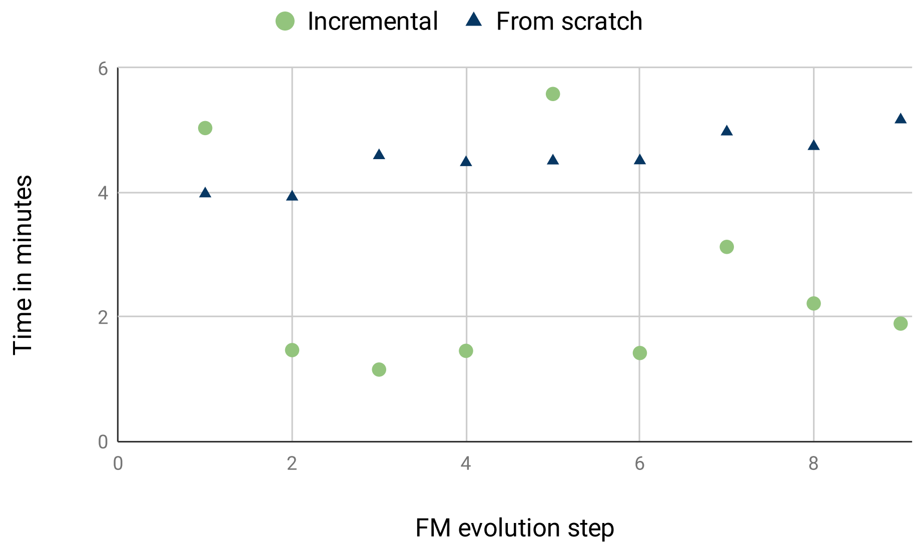


Figure 5.12: Calculation times for all FinancialServices01 evolution steps for a complete MIG without affected clauses.

the calculation times needed when finding affected clauses, we can see that the calculation without affected clauses takes more time. Additionally, this method is not more accurate than calculating with affected clauses. When evaluating this calculation with Automotive02 we received a warning and, e.g., Automotive02_V2 to Automotive02_V3 took 380 minutes to calculate. Since we did not have the capacity to find the cause of the warning and the results of this calculation are not very promising anyway, we did not evaluate the calculation for complete MIGs without affected clauses further.

RQ2

As SAT solver calls are very expensive and a driving factor for the complexity of the MIG calculation, we analyze in RQ2 to which extent we are able to reduce necessary SAT solver calls for each calculation we presented in Section 5.2. We distinguish between the SAT calls necessary for the calculation of (1) redundant clauses (i.e., an incomplete MIG with redundant clauses) and (2) implicit strong edges (i.e., a complete MIG without redundant clauses). Without redundant clauses and implicit strong edges, we only call a SAT solver when calculating dead and core features. Since the algorithms for this calculation are the same for the incremental constructed MIG and the MIG calculated from scratch, we do not have to evaluate the number of SAT calls in this case.

With redundant clauses calculation

First, we analyze the number of SAT calls that are required for detecting redundant clauses when (incrementally) calculating the MIG. Figure 5.13 shows the number of SAT calls necessary to find the redundant clauses of each BusyBox FM. The diagram shows that, except one outlier, the incremental constructed MIG needs less time than the MIG calculated from scratch. Comparing the number of SAT calls to Figure 5.4, which shows the computation time of an incomplete graph with redundant clauses, we can see similarities between the computation times and the number of SAT calls.

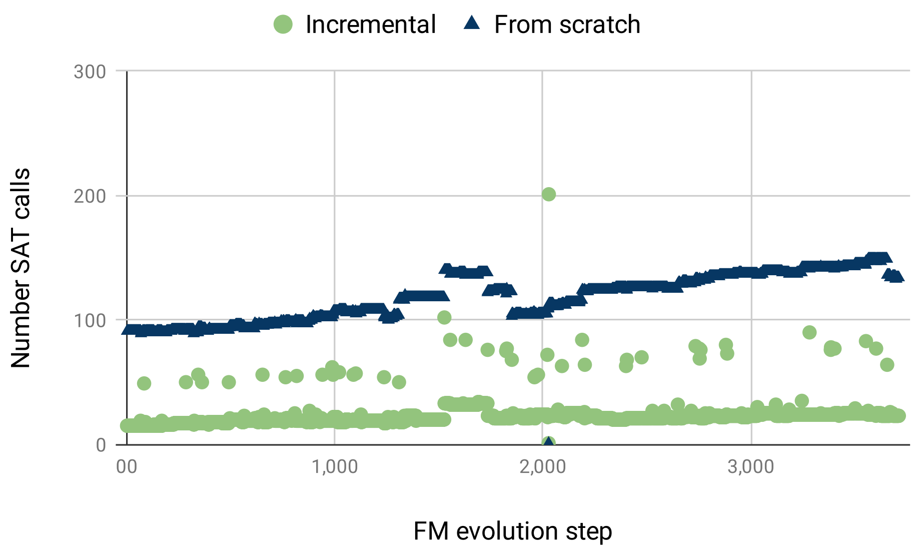


Figure 5.13: Number of SAT calls necessary for all BusyBox evolution steps for the calculation of redundant clauses.

Figure 5.14 shows the SAT calls of the FinancialServices01 FMs. Again, the diagram shows that, except for two outliers, the incremental construction requires less SAT calls than the computation from scratch. The diagram shows, that the difference between the two calculations is not large (i.e., the incremental construction requires 8,841 SAT calls and the construction from scratch requires 9,070 SAT calls for the eight evolution steps). But the number of SAT calls necessary for the redundant clauses calculation is relatively small anyway.

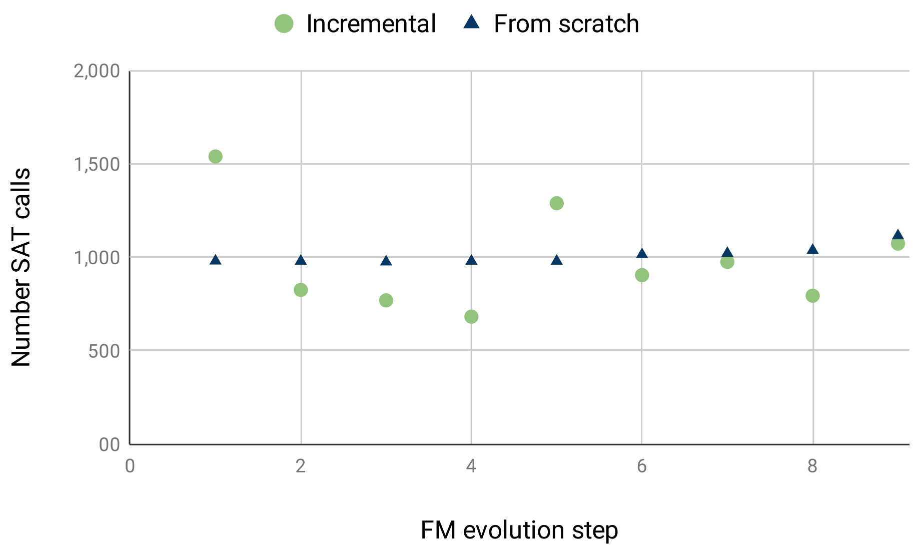


Figure 5.14: Number of SAT calls necessary for all FinancialServices01 evolution steps for the calculation of redundant clauses.

Now, we examine the required SAT calls for redundant clauses for the Automotive02 FM evolution steps, displayed in Figure 5.15. We can see that the incremental construction needs a few less SAT calls for the second and third evolution step. In

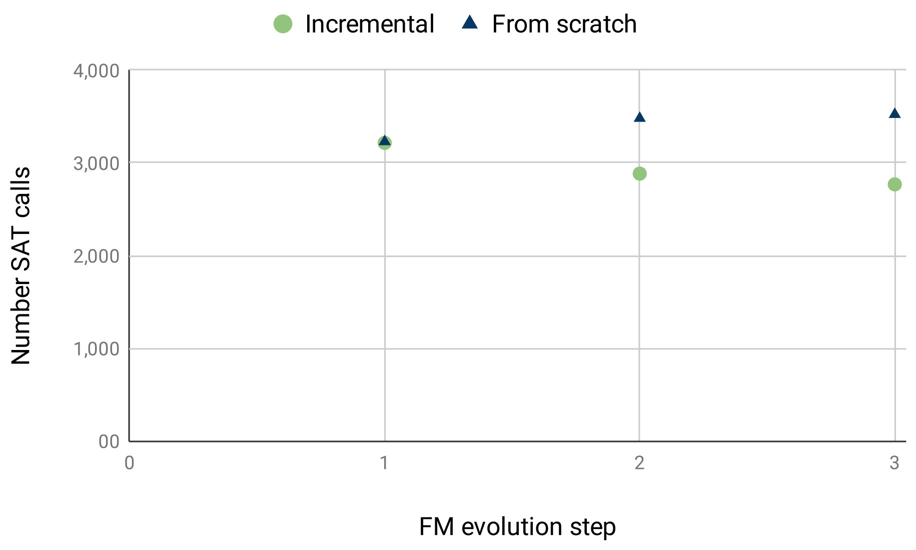


Figure 5.15: Number of SAT calls necessary for all Automotive02 evolution steps for the calculation of redundant clauses.

the first, both calculation need about the same number of SAT calls (3,216 for incremental vs. 3,229 for from scratch). Overall, similar to the number of SAT calls necessary for the FinancialServices01 models, the incremental calculation does not need significantly less SAT calls than the calculation from scratch.

With implicit strong edges

Second, we inspect the number of SAT calls that both MIG calculations need for the detection of implicit strong edges. Figure 5.16 shows the number of SAT calls necessary for all BusyBox FM evolution steps. We can see that the incremental construction needs much fewer SAT calls than the calculation from scratch. On the average, the incrementally calculated MIG calls a SAT solver 51.87 times and the calculation from scratch 2,980.25 times.

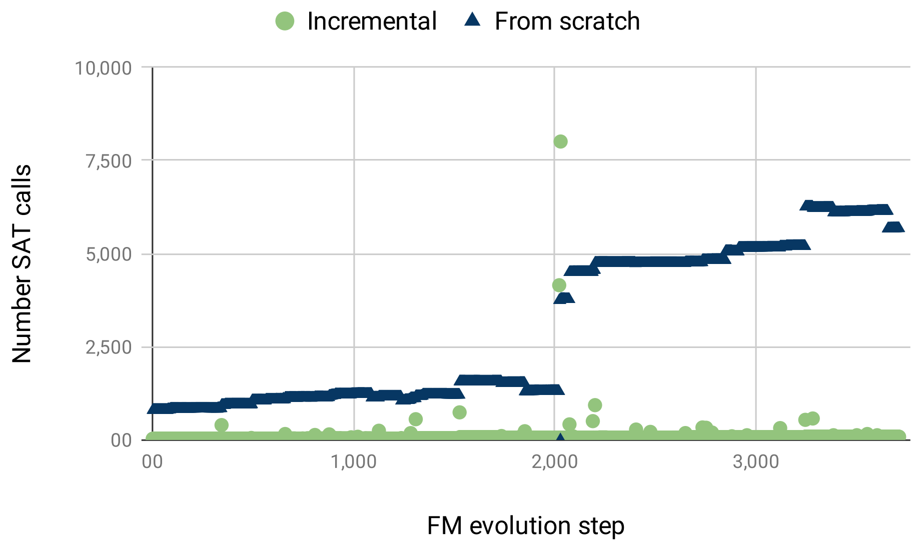


Figure 5.16: Number of SAT calls necessary for all BusyBox evolution steps for the calculation of implicit strong edges.

Figure 5.17 shows the SAT calls necessary in the calculation for all FinancialServices01 evolution steps. When comparing to the computation time necessary in this specific calculation (cf. Figure 5.9), we can see a correlation between the number of SAT calls and the necessary computation time.

The last FMs we evaluate are the Automotive02 versions in Figure 5.18. On the one hand, we can see that the number of SAT calls necessary for the incremental construction in the first evolution step is more than 4 times as large as for the calculation from scratch. On the other hand, the SAT calls for the second and third evolution step are more than 4 times and more than 7 times larger for the calculation from scratch.

RQ3

Finally, we investigate the differences between the incrementally constructed MIG and the MIG calculated from scratch. Even though the MIGs are both logically correct, we analyze their difference regarding the number of (1) detected redundant

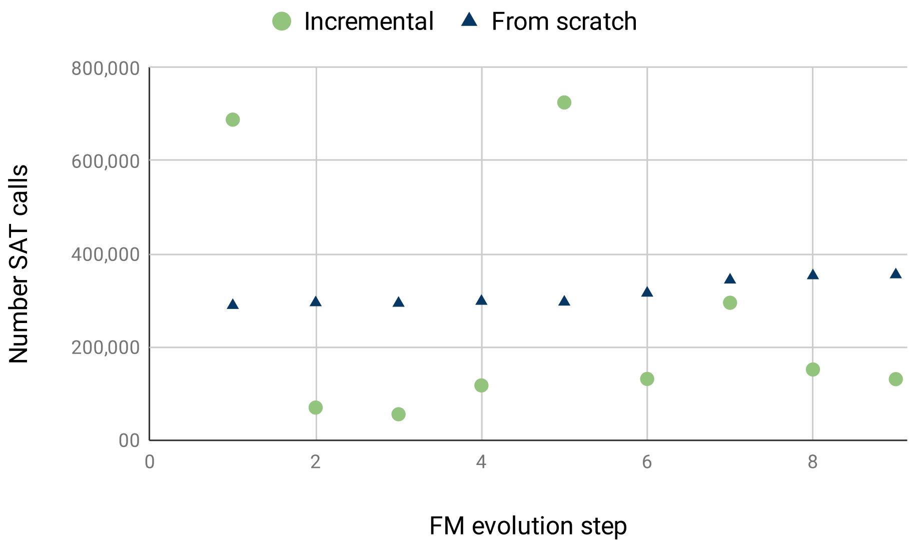


Figure 5.17: Number of SAT calls necessary for all FinancialServices01 evolution steps for the calculation of implicit strong edges.

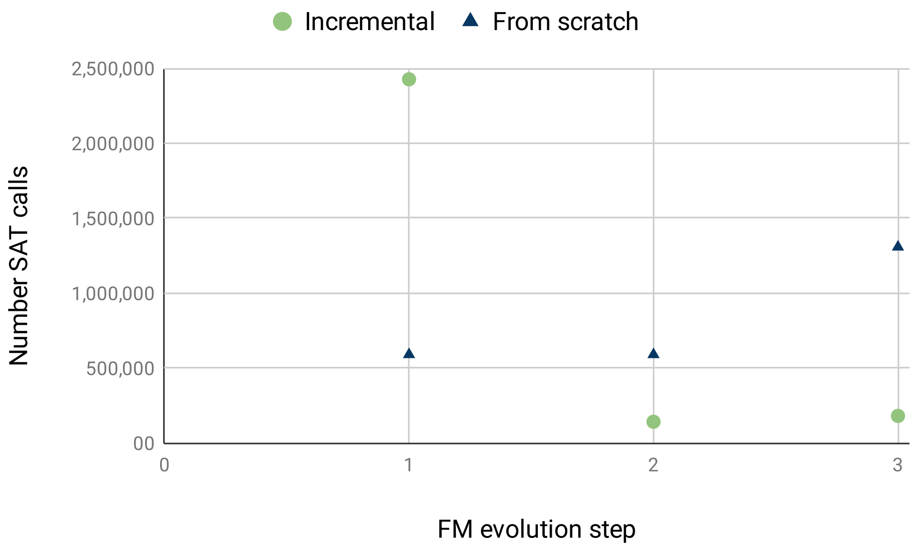


Figure 5.18: Number of SAT calls necessary for all Automotive02 evolution steps for the calculation of implicit strong edges.

clauses and (2) implicit strong edges. This way, we inspect the degree of accuracy of the incremental constructed MIG in comparison to the MIG calculated from scratch in order to analyze the trade-off between the accuracy and the computation time between the incremental construction and the calculation from scratch.

Redundant Clauses Difference

For BusyBox we have 3,712 FM versions. For better readability, we present the sum of the difference of the redundant clauses for all 3,712 steps and not the difference for each single step. In total, the from scratch calculated MIG detected 190 redundant clauses that the incremental calculation did not detect. The incremental calculation detected 184 redundant clauses that the calculation from scratch did not find.

For FinancialServices01 we have ten FM versions with nine evolution steps. In Figure 5.19, we show the difference between the number of detected redundant clauses for each evolution step. As the diagram depicts, the incremental calculation finds more redundant clauses for each evolution step.

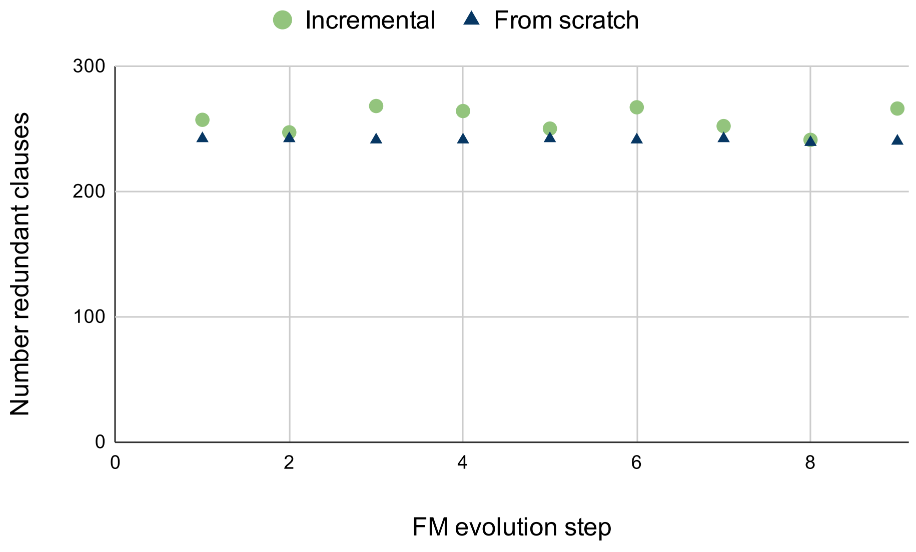


Figure 5.19: Number of found redundant clauses for all FinancialServices01 MIGs.

Figure 5.20 shows the redundant clauses that both kinds of calculation have detected. The diagram shows that the incremental construction and the calculation from scratch find around the same number of redundant clauses.

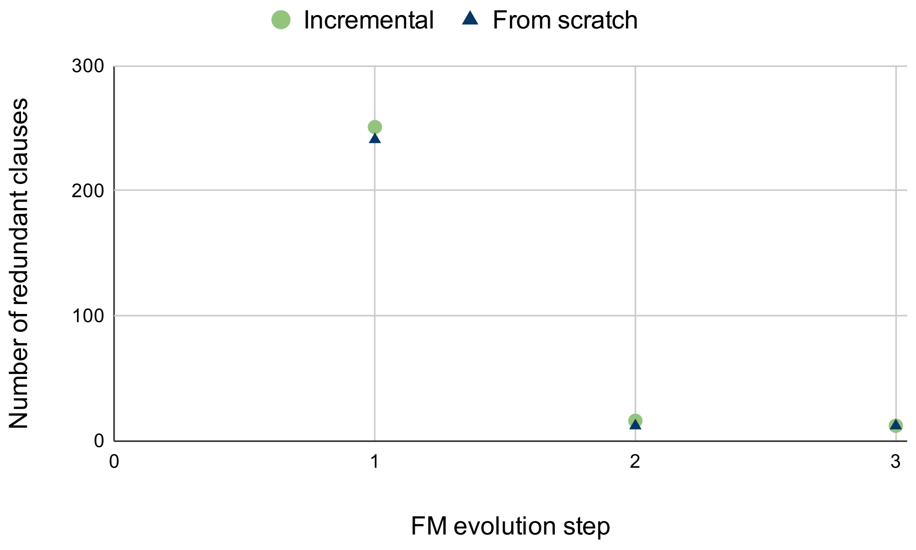


Figure 5.20: Number of found redundant clauses for all Automotive02 MIGs.

Implicit Strong Edges Difference

For the BusyBox models, we evaluated all 3,712 FM versions as one. We detected differences in the implicit strong edges in only 2 of the 3,711 evolution steps. This means, that the incremental construction found almost every implicit strong edge that the calculation from scratch detects.

Now, we show the difference in the implicit strong edges for the FinancialServices01 MIGs in Figure 5.21. As the diagram shows, the incremental calculation finds almost as many implicit strong edges as the calculation from scratch (i.e., the incremental calculation finds 359,946 and the calculation from scratch finds 363,333 in total).

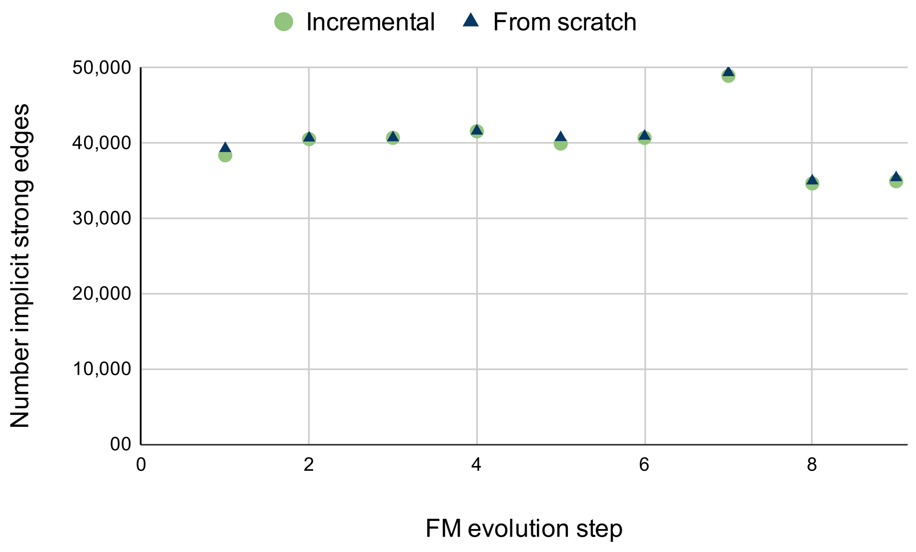


Figure 5.21: Number of found implicit strong edges for all FinancialServices01 MIGs.

For the Automotive02 MIGs, the difference regarding the implicit strong edges, depicted by the diagram in Figure 5.22, is zero for the first evolution step (both calculations found 166 implicit strong edges). In the second evolution step, the incremental calculation found 634 implicit strong edges while the calculation from scratch found 700. In the third evolution step, the incremental calculation found 690 implicit strong edges and the calculation from scratch 696.

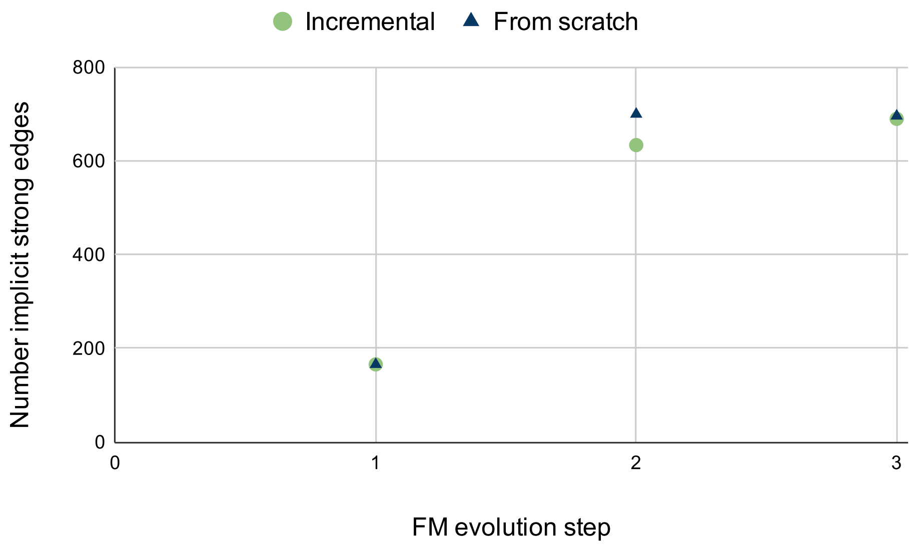


Figure 5.22: Number of found implicit strong edges for all Automotive02 MIGs.

5.3 Discussion

Regarding the RQ1, the evaluation shows that for most cases, the incremental calculation of the MIG takes less time than calculating the MIG from scratch. As Table 5.3 and Table 5.4 show, when constructing an incomplete MIG, the most time is saved during the calculation of transitive strong edges. Since this calculation is the same as for the from scratch calculated MIG except a few adaptions, it is only faster because of the way we implemented the method and not because of its incrementality. When disregarding the time for this step, Table 5.3 and Table 5.4 show that the incremental calculation still requires less computation time.

When constructing a complete MIG, depending on the size of the changes, the incremental calculation brings a large advantage. When inspecting the necessary calculation times for Automotive02, the first evolution step takes the incremental construction a lot of more time than for the other evolution steps. However, the first evolution step includes large changes in the CNF (i.e., more clauses have been added than clauses contained in the CNF before the evolution step). The second and third evolution steps also include relatively large changes to the FM, but the incremental construction is faster anyway. In summary we can say that the incremental construction brings a bigger advantage for large-scale feature models regarding the computation time. Also, the bigger the changes to the FM and, thus, to the CNF, the less advantage brings the incremental construction. This claim is also supported by Figure 5.9 where the first and the fifth evolution step are the only ones were the incremental construction takes more time than calculating from scratch. Table 5.2 shows, that these are by far the largest changes in the CNF for all nine evolution steps as there are over 2,000 added clauses whereas the other steps have less than 700 added clauses.

When evaluating RQ2 and, thus, the necessary SAT calls for every calculation, we can see that there is a connection between the calculation time and number of SAT calls in the calculation. This confirms our assumption that the reduction of SAT calls is an effective approach for reducing the calculation time which we achieved with our incremental calculations. For the vast majority of the evolution steps that we evaluated, the incremental construction requires fewer SAT calls than the calculation from scratch.

We detected that the incremental calculation of redundant clauses needs less time without a high loss of accuracy as we can see by the fact that it finds more redundant clauses in most of the cases we evaluated in RQ3. The reason is how the SAT solver works. Every clause that is already negatively checked for redundancy is added to the SAT solver. Clauses that were not checked yet are not part of the SAT solver. It might be that a clause that would be redundant is checked before the set of clauses causing the clause to be redundant is part of the SAT solver. Hence, the clause would not be declared redundant. In the incremental construction, the clauses that are not directly affected by one of the added clauses are added to the SAT solver before we inspect the potentially redundant clauses. Since it is more likely that clauses that share a literal with one of the the added clauses or one of the added clauses are also affected in their redundancy, the incremental construction enlarges the chance to find them due to the already existing set of remaining clauses in the SAT solver. Consider the following example: we have the clauses $\{B,\ C\}$ and $\{A,\ \neg C\}$. These clauses express, that when deselecting C we have to select B and when selecting C we have to select A. Then, we add the clause $\{A,\ B\}$ which is redundant, because either A or B have to be selected due to the already existing clauses. When calculating from scratch, we might check the clause $\{A,\ B\}$ first and since the other clauses are not checked yet and, thus, are not part of the CNF in the SAT solver yet, we would not detect the redundancy of the clause $\{A,\ B\}$. In the incremental calculation, the clauses $\{B,\ C\}$ and $A,\ \neg C$ are part of the SAT solver before verifying $\{A,\ B\}$, since the added clauses are checked last with all the remaining clauses already in the CNF of the SAT solver. Thus, the incremental calculation would detect the redundancy of the specific clause.

As we also show when answering RQ3, the incremental construction does not find all implicit strong edges. The main cause of implicit strong edges are cross-tree constraints, since they cause relations between features that are neither expressed by the FM nor explicitly by a constraint. For the FinancialServices01 evolution steps, where the incremental construction misses the most implicit strong edges, we argue that the loss is relatively small compared to the time we can save. Comparing the results of FinancialServices01 and Automotive02 it might seem like an irregularity that the incremental construction misses for Automotive02, which has much more added and removed clauses, less implicit strong edges. This is probably due to the fact that FinancialServices01 has more cross-tree constraints in relative terms. While the Automotive02 versions have between 14,010 and 18,616 features and between 666 and 1,319 cross-tree constraints, FinancialServices01 has around 700 features but around 1,000 cross-tree constraints. That means, that FinancialServices01 has a lot of features that have a relation due to the cross-tree constraints and, hence, the MIGs of these FMs have a lot more implicit strong edges than the MIGs of the Automotive02 FM versions. Anyway, missing implicit strong edges has no impact on the correctness of the MIG but on the performance during the configuration construction. The implicit strong edges give a small benefit regarding the time needed to find the connections between features and, thus, the consequences of a (de-)selection of a feature.

The evaluation shows that the efficiency of the incremental calculation depends on multiple characteristics of the FM (i.e., the size, the number of changes, and the relative number of cross-tree constraints). The advantage of the incremental construction grows with the size of the feature models. The reason is, that the required time for the calculation from scratch typically grows with the size of a FM. When adapting the MIG to changes the size of the feature model has much less influence on the computation time than the number of changes that have to be adapted. The relative number of cross-tree constraints also influences the incremental adaption in a way that more cross-tree constraints lead to more implicit strong edges. Anyway, the goal of this thesis was to reduce the high number of SAT calls and the related high computation time when constructing a MIG and still have a resulting MIG that is efficient and correct. The evaluation strongly indicates that the goal is achieved. Even though we do not have a fully complete MIG resulting by the incremental construction, the loss is relatively small when comparing to the time we can save with the introduced concept.

5.4 Threats to Validity

In this section, we describe the threats to the validity of our evaluation.

We cannot guarantee the efficiency of the incremental construction of a MIG when using for FMs we did not evaluate in this thesis. To prevent that and give the best overview of the advantage of the incremental construction, we evaluated FMs from multiple domains with different sizes and FM evolution steps with a different number of changes in the FM. To present the best use cases for the incrementally calculated MIG we identified the characteristics of FMs that bring advantages for the incremental construction in Section 5.3.

We did not formally prove the correctness of the incrementally constructed MIG. Thus, we cannot completely guarantee the correctness of the incrementally constructed MIG. To this end, we made a lot of sanity checks. For example, we compared the resulting MIGs not only by time but also each variable to verify that the MIGs have the same strong edges, the same weak edges, and the same attributes (i.e., the same variables are core/dead). When detecting differences, like a clause that exists in the from scratch calculated MIG but not in the incrementally constructed MIG, we verified that it is a redundant clause and not a falsely ignored clause.

When using Java for the implementation, just-in-time compilation can have an impact on the required time for each calculation. To prevent the calculation times to be influenced by the JVM, we run a JVM warm up before each calculation.

We did not evaluate the additional memory that is required when incrementally calculating. Since we have to save a lot more information than necessary for the calculation from scratch, this might be an interesting part to evaluate in the future. However, for large-scale feature models this can be outsourced on servers. The money for the server can be retrieved by the time savings of the incremental construction.

6. Related Work

In this chapter, we describe work that is related to this thesis. We discuss feature model evolution and incremental SAT solvers.

Multiple researchers investigate feature model evolution on a feature level. Pleuss et al. [PBD+12] split FMs into *fragments* wherein changes in the fragments due to evolution are modeled by *EvoOperators*. Cordy et al. [CCS+12] analyze feature model changes whereas they reduce the model with knowledge about the edits for model checking after evolution steps. Thüm et al. [TBK09] divide edits to feature models into four classifications. Even though they also use the CNF of feature models to verify changes in the FM, that is done only for the classifications and not for finding the exact difference in the CNFs of the FMs as base for FM evolution analysis. Hoff et al. [HNS+20] save FM evolution operations and determine changes in the FM by inspecting the performed operations. The described information about FM evolution operations could be used for future work to incrementally adapt MIGs more efficiently. Neves et al. [NTS+11] introduce multiple templates for the analysis of different FM evolution scenarios. However, none of these approaches uses CNF differences for feature model evolution changes. Acher et al. [AHC+12] introduce a concept of composing and decomposing FMs. They determine the FM differences of two FMs by translating them into a CNF and into a binary directed graph and compute the FM difference with the CNF or with the graph.

The literature contains multiple approaches for incremental SAT solvers [ES03, FBS19, NRS14]. Besides, Een et al. [EB05] introduce techniques to reduce a CNF to speed up the usage of a SAT solver. However, since the goal of this thesis is to analyze the advantage of the incremental MIG compared to the MIG calculated from scratch, we did not consider improvements regarding the SAT solver and instead adapted the SAT solver that already exists in the implementation of a MIG. Investigating the usage of incremental SAT solvers for improvements of the incremental calculation of a MIG is an interesting factor for future work.

https://doi.org/10.24355/dbbs.084-202107050823-0

7. Conclusion and Future Work

In this thesis, we introduced all concepts that are necessary for the incremental calculation of a MIG and provided an implementation realizing our method. We also presented an evaluation of our work for multiple industrial feature models with different evolution steps. For several steps of the incremental construction, we evaluated multiple algorithms to examine trade-offs between accuracy and runtime. In summary, we conclude that on the one hand the incrementally constructed MIG is more efficient regarding the computation time and the reduction of number of necessary SAT calls. On the other hand, it has a small loss of accuracy regarding the redundant clauses and the implicit strong edges. We argue that this loss is relatively small compared to the possible reduction of computation time and that the incremental construction is especially worth using when calculating a complete MIG. Our empirical evaluation indicates that the advantage grows for larger feature models when incrementally adapting the MIG after every evolution step in the FM. However, even for small feature models the incremental calculation is typically more efficient. Thus, we argue that applying the incremental construction yields benefits for feature models of various sizes. In general, the improvement regarding efficiency is larger for the calculation of a complete MIG since we save the most time when detecting implicit strong edges.

We identified multiple possible improvements for the incremental construction as future work. First, we can analyze whether we can use the knowledge of the previously dead and core features for an incremental algorithm of the calculation of the new dead and core features. Second, we can investigate changes on the level of FM edit operations (e.g., move feature) to detect affected features and, thus affected clauses more efficiently. This way, we might ignore clauses that are unnecessarily observed. Third, we can improve the search for implicit strong edges to save even more SAT calls. The MIG calculated from scratch uses a depth first search and a method for a preselection involved in the calculation of implicit strong edges which we could adapt for the incremental calculation. Thereupon, we can analyze whether the knowledge of changes in the MIG can be used to fix existing configurations that are invalid after the evolution of its FM.

A. Appendix

Figure A.1 shows the calculation for the FinancialServices01 models for complete graphs without redundant clauses.

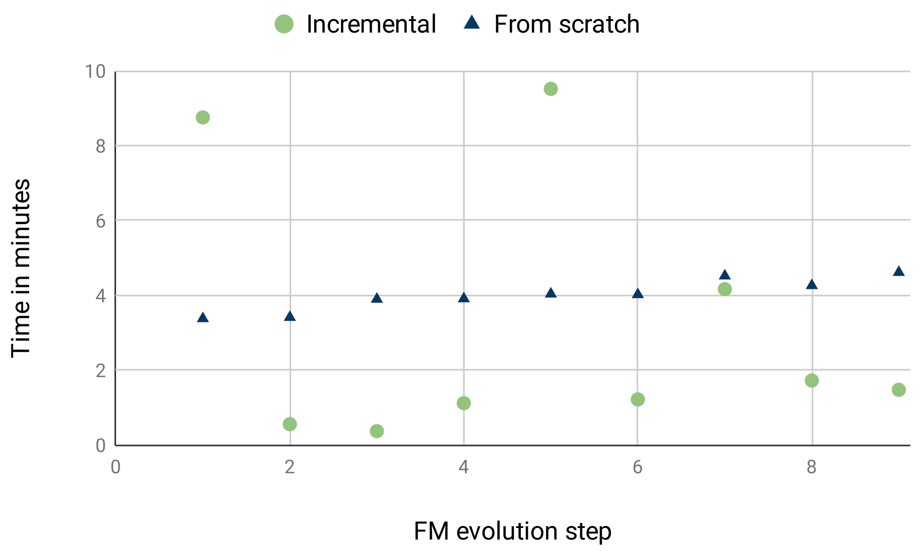


Figure A.1: Calculation times for all FinancialServices01 evolution steps for a complete MIG without redundant clauses.

Figure A.2 shows the calculation for the Automotive02 models for complete graphs without redundant clauses.

Table A.1 shows the exact times for each calculation for the evolution step from Automotive02_V2 to Automotive02_V3 for a complete graph without redundant clauses.

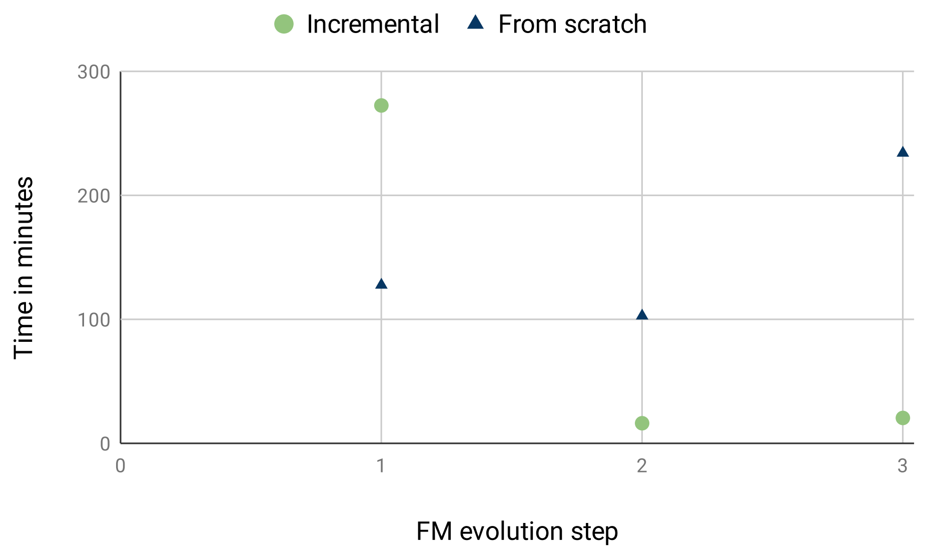


Figure A.2: Calculation times for all Automotive02 evolution steps for a complete MIG without redundant clauses.

Calculation Step	Calculated MIG (sec)	Adapted MIG (sec)	Time Difference Adapted MIG - Calculated MIG(sec)
Remove old transitive strong edges	0	0.09	+0.09
Calculate core and dead feature	8.92	8.94	+0.02
Clean CNF	0.18	0.1	-0.08
Calculate CNF difference	0	0.48	+0.48
Removed clauses	0	0.01	+0.01
Added clauses	0	2.05	+2.05
Find transitive strong edges	33.11	5.54	-27.57
Find transitive weak edges	12.65	0.8	-11.85
Find implicit strong edges	5,958.84	974.57	-4,984.27
Create MIG	1.86	0	-1.86
Calculate redundant clauses	0.02	0	-0.02
Find transitive strong edges	32.95	0.04	-32.91
Overall time	6,048.71	992.83	-5,055.88

Table A.1: Calculation times for each step for the complete MIG without redundant clauses in the evolution step from Automotive02_02 to Automotive02_03.

Bibliography

[AHC+12] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. Feature model differences. In *International Conference on Advanced Information Systems Engineering*, pages 629–645. Springer, 2012. (cited on Page 49)

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005. (cited on Page 1, 3, and 4)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information systems*, 35(6):615–636, 2010. (cited on Page 1 and 3)

[CCS+12] Maxime Cordy, Andreas Classen, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Managing evolution in software product lines: A model-checking perspective. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, pages 183–191, 2012. (cited on Page 49)

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, 10(1):7–29, 2005. (cited on Page 1 and 3)

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *11th International Software Product Line Conference (SPLC 2007)*, pages 23–34. IEEE, 2007. (cited on Page 1 and 3)

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *International conference on theory and applications of satisfiability testing*, pages 61–75. Springer, 2005. (cited on Page 49)

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003. (cited on Page 49)

[FBS19] Katalin Fazekas, Armin Biere, and Christoph Scholl. Incremental inprocessing in sat solving. In *International Conference on Theory and Appli-*

cations of Satisfiability Testing, pages 136–154. Springer, 2019. (cited on Page 49)

[HNS$^+$20] Adrian Hoff, Michael Nieke, Christoph Seidl, Eirik Halvard Sæther, Ida Sandberg Motzfeldt, Crystal Chang Din, Ingrid Chieh Yu, and Ina Schaefer. Consistency-preserving evolution planning on feature models. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, pages 1–12, 2020. (cited on Page 49)

[IF10] Ayelet Israeli and Dror G. Feitelson. The Linux kernel as a case study in software evolution. 83(3):485–501, 2010. (cited on Page 1)

[Jan08] Mikolas Janota. Do sat solvers make good configurators? In *SPLC (2)*, pages 191–195, 2008. (cited on Page 5)

[Kri15] Sebastian Krieter. *Efficient Configuration of Large-Scale Feature Models Using Extended Implication Graphs.* PhD thesis, Citeseer, 2015. (cited on Page 5)

[KTS$^+$09] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented software development. In *2009 IEEE 31st International Conference on Software Engineering*, pages 611–614. IEEE, 2009. (cited on Page 21)

[KTS$^+$18] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. Propagating configuration decisions with modal implication graphs. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 898–909. IEEE, 2018. (cited on Page 1, 2, 5, 9, and 18)

[KZK10] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. Model counting in product configuration. *arXiv preprint arXiv:1007.1024*, 2010. (cited on Page 1 and 4)

[MTS$^+$17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering software variability with FeatureIDE.* Springer, 2017. (cited on Page 21)

[NRS14] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Ultimately incremental sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 206–218. Springer, 2014. (cited on Page 49)

[NTS$^+$11] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulesza, and Paulo Borba. Investigating the safe evolution of software product lines. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, pages 33–42, 2011. (cited on Page 49)

[PBD$^+$12] Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. Model-driven support for product line evolution on feature level. *Journal of Systems and Software*, 85(10):2261–2274, 2012. (cited on Page 49)

[SSK$^+$20] Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. SMT-Based Variability Analyses in FeatureIDE. February 2020. (cited on Page 1)

[STS20] Chico Sundermann, Thomas Thüm, and Ina Schaefer. Evaluating #sat solvers on industrial feature models. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 1–9, 2020. (cited on Page 1 and 4)

[TBK09] Thomas Thum, Don Batory, and Christian Kastner. Reasoning about edits to feature models. In *2009 IEEE 31st International Conference on Software Engineering*, pages 254–264. IEEE, 2009. (cited on Page 49)

[TKB$^+$14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014. (cited on Page 2 and 21)

Task

Many software systems nowadays have a large number of features with a much larger number of different variants, e.g., the operating system of a mobile phone. A software product line (SPL) is used to manage the features and to provide reusability of their functionalities. For a better overview on the features, a feature model (FM) organizes them hierarchically in a tree structure that describes the dependencies between the features. With an FM, all valid combinations of features, called configurations, are defined, and a configuration's validity can be tested using the FM. The logic of an FM can be represented by a Boolean formula.
However, when selecting specific features while constructing an individual configuration, it is possible that other features become either implicitly selected or deselected due to constraints in the FM. To prevent the user from selecting an invalid combination of features, the configuration validity needs to be tested during this process, which is called the *online phase*. Validating the configuration during the online phase to inform the user about resulting consequences when selecting or deselecting features is called *decision propagation*. The validity check can be done by a SAT solver, that checks whether a Boolean term is satisfiable. Calling a SAT solver each time a feature gets selected or deselected to check which other features are still valid to be selected costs a lot of time in the decision propagation. To improve this situation, Modal Implication Graphs (MIGs) have been devised that can be used to identify implicit selection during the configuration process. The MIG is computed only once in the *offline phase*, to be used afterwards in the online phase.
A conjunctive normal form (CNF) as a logical representation of the FM serves as base for the computation of a MIG. A CNF describes a FM as a logical formula and contains a number of clauses that consist of literals. A clause represents the dependency between features by connecting the literals with a logical OR. Each literal represents the selection or deselection of a feature (e.g., having two features A and B with A↔B, the clause would be (¬A, B)). All clauses are connected with a logical AND. A MIG consists of nodes, where each node represents a literal from the CNF. The nodes are connected by strong and weak edges that represent the connection between the features. Therefore, clauses with only one literal are ignored, a clause with two literals is represented by a strong edge and a clause with more than two literals by a weak edge. Despite the fact that a lot of time is saved during the configuration process, constructing the MIG in the first place takes a huge amount of time. Changing the FM in any way results in a changed CNF and, thus, requires a recalculation of the respective MIG. This is even necessary if the changes to the FM are small. Hence, a technique to incrementally compute a MIG based on FM changes could make the use of MIGs more efficient.

Introducing an algorithm for the incremental computation of a MIG

In this thesis, I will provide a method to construct an incremental MIG for an evolving FM. I will use the changes in the resulting CNF and compute their impact on the respective MIG. As basis for the incremental algorithm, I calculate the difference between the CNF before and after the changes. During the construction of an algorithm for the computation, I differentiate between complete and incomplete MIGs. An incomplete MIG does not contain all possible strong edges but is correct anyway. A complete MIG cannot be completely derived by the CNF, but requires additional computations by a SAT solver. Therefore, I will focus on incomplete MIGs in the first place. In the end, I will analyze the impact of the feature-model evolution operations (e.g., create or move a feature) on the CNF and investigate on how to improve the computation time of my algorithm.
To evaluate my method, I will test the correctness of the resulting graph. To do so, I will use small samples, generate the MIG, and evaluate the result. In addition, I will compare the time my algorithm needs to adapt the MIG with the time the original algorithm needs to recalculate the entire MIG. To this end, I will use different sizes of FMs as well as different numbers of changes that have been made since the last calculation. I will also compare the resulting MIGs by their size to inspect that the adapted MIG is not significantly larger than the newly calculated one.

For the development process I consider the following work packages:
Work package 1: Literature Study. I will study research on informations about *"Software Product Lines"*, *"Modal Implication Graphs"*, *"Differencing Algorithms"*, *"SAT Solver"*
Work package 2: Concept development. I subdivide the second work package in several smaller tasks:

- The first package is to find a concept for the incomplete graph to adapt. Therefore, I will develop an algorithm that detects the difference between the CNF before and after changes in the FM and adapts the MIG. I will do so for incomplete MIGs.
- In the second package, I need to analyze the impact of changes in the FM on the CNF and, accordingly, on the MIG.
- In the third package, I will investigate the possibility to extend the concept for complete graphs.
- The fourth package is to investigate whether knowing what operations have been made can simplify the process in some cases.

Work package 3: Implementation. This includes implementing the aforementioned algorithm in FeatureIDE, where it will be used when a MIG needs to be recalculated.
Work package 4: Evaluation. I will examine the correctness of my method as well as its computation time and the size of the resulting MIG.
Work package 5: Writing of the thesis.

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Braunschweig, den 08. Januar 2021

https://doi.org/10.24355/dbbs.084-202107050823-0