Incremental Construction of Modal Implication Graphs for Evolving Feature Models

Sebastian Krieter
Harz University of Applied Sciences
Wernigerode, Germany

Rahel Arens
TU Braunschweig
Brunswick, Germany
University of Ulm
Ulm, Germany

Michael Nieke
ITU Copenhagen
Copenhagen, Denmark
TU Braunschweig
Brunswick, Germany

Chico Sundermann
Tobias Heß
Thomas Thüm
University of Ulm
Ulm, Germany

Christoph Seidl
ITU Copenhagen
Copenhagen, Denmark

ABSTRACT

A feature model represents a set of variants as configurable features and dependencies between them. During variant configuration, (de)selection of a feature may entail that other features must or cannot be selected. A Modal Implication Graph (MIG) enables efficient decision propagation to perform automatic (de)selection of subsequent features. In addition, it facilitates other configuration-related activities such as t-wise sampling. Evolution of a feature model may change its configuration logic, thereby invalidating an existing MIG and forcing a full recomputation. However, repeated recomputation of a MIG is expensive, and thus hampers the overall usefulness of MIGs for frequently evolving feature models. In this paper, we devise a method to incrementally compute updated MIGs after feature model evolution. We identify expensive steps in the MIG construction algorithm, enable them for incremental computation, and measure performance compared to a full rebuild of a complete MIG within the evolution histories of four real-world feature models. Results show that our incremental method can increase the speed of MIG construction by orders of magnitude, depending on the given scenario and extent of evolutionary changes.

KEYWORDS

Configurable System, Software Product Line, Feature Modeling, Evolution, Configuration

ACM Reference Format:
Sebastian Krieter, Rahel Arens, Michael Nieke, Chico Sundermann, Tobias Heß, Thomas Thüm, and Christoph Seidl. 2021. Incremental Construction of Modal Implication Graphs for Evolving Feature Models . In *Proceedings of 25th ACM International Systems and Software Product Lines Conference (SPLC'21).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SPLC'21, 06–11 September, 2021, Leicester, UK

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A *Software Product Line (SPL)* captures a family of closely related software *variants* [1, 7, 9, 28]. On a conceptual level, a variant is defined via a *configuration* comprised of configuration options, i.e., selected or deselected *features* [1, 7, 9, 28]. A *feature model* captures features of an SPL and their relations, such as implications and exclusions, as *constraints* [1, 7, 9, 28]. Real-world feature models commonly grow large, resulting in a massive number of features and complex constraints [4, 5, 30]. As a consequence, defining a valid configuration is challenging for engineers, because they have to obey all constraints when (de)selecting features. Related applications, such as configuration sampling [8, 16, 19, 32], suffer from similar challenges to derive and reason on (many) valid configurations. To support the configuration process, a *Modal Implication Graph (MIG)* facilitates efficient *decision propagation* by directly modeling the impact of feature (de)selections and automatically (de)selecting subsequent features [16, 17]. While a generated MIG can be reused to support an unlimited number of configuration processes, it has to be specifically tailored to encode the configuration logic of a particular feature model, which entails significant computational cost. Feature model evolution may change a feature model or its constraints and, subsequently, invalidate an existing MIG that then represents outdated configuration logic. For feature models that frequently evolve, it is costly to perform a full rebuild of a complete MIG after each evolution step [17]. Thus, in the light of frequent feature model evolution, reaping the benefits of a MIG for automating part of the configuration process or sampling configurations is, currently, severely hampered if not outright infeasible.

In this paper, we present a method to incrementally create a MIG for an evolving feature model. After feature-model evolution, we reuse information from a previously built MIG and compute the impact of the feature-model changes on the MIG. To this end, we identify which steps of the original MIG creation algorithm are the most costly and can benefit from incremental computation. We define an overall incremental build process for MIGs consisting of

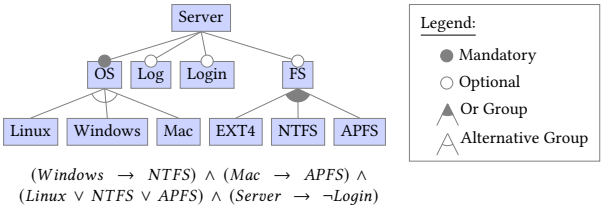

Figure 1: Example feature model `Server`

steps from the original build process as well as incremental steps that reuse information from a previous MIG. Thus, we enable efficient creation of MIGs in presence of feature-model evolution. Our method makes the benefits of MIGs accessible even for frequently changing feature models. We evaluate our method by comparing the incremental MIG creation with the original build process. To this end, we analyze improvements in performance when using the incremental construction for the feature models of four real-world feature models from different domains with different sizes, complexity, and evolution frequency.

In summary, we contribute the following:

- A method for building a Modal Implication Graph (MIG) incrementally in accordance with feature-model evolution.
- An open-source implementation for incremental MIG generation based on FeatureIDE[1].
- An evaluation of the performance of incremental MIG creation on four real-world systems.
- A nuanced recommendation for when the incremental build process provides a performance improvement depending on characteristics of a feature model and its evolution.

With our contributions, we enable engineers to make use of the benefits provided by MIGs, even in presence of frequently changing feature models. We show that the incremental build process can be up to 100 times faster than computing a new MIG depending on the application scenario. The results of our evaluation show that incremental MIGs are always similarly effective than original MIGs when utilizing them in decision propagation. Based on these findings, we give advice for engineers in which scenarios the incremental creation of MIGs is faster and when to use the original build process.

2 BACKGROUND

In the following, we present our notion of feature models, configurations, and Modal Implication Graphs.

2.1 Feature Models

A *feature model* $M = (F, C)$ defines the configuration space of an SPL. It consists of a set of features $F = \{f_1, ..., f_n\}$ and a set of constraints $C = \{c_1, ..., c_m\}$ over these features. A *feature* is a Boolean variable that can be either *selected* or *deselected* in a configuration of an SPL. A *constraint* is a propositional formula over the set of features that limits the space of valid configurations. Each constraint consists of a propositional formula over the set of features.

[1]https://featureide.github.io/

In Figure 1, we show an example feature model for a simple server system represented as a feature diagram. The feature model defines the features `Server`, `OS`, `Log`, `Login`, `FS`, `Linux`, `Windows`, `Mac`, `EXT4`, `NTFS`, and `APFS`. The tree structure of the feature diagram and any additional cross-tree constraints below the feature tree define the set of constraints. For instance, the selection of the feature `FS` implies that at least one of the features `EXT4`, `NTFS`, or `APFS` must be selected as well (i.e., an OR-group). This can be expressed as the constraint $FS \rightarrow (EXT4 \vee NTFS \vee APFS)$. The features `Server` and `OS` are mandatory and must be selected. The features `Linux`, `Windows`, and `Mac` are in an ALTERNATIVE-group and thus exactly one of them must be selected.

Most automated reasoning techniques for feature models, including the creation of a MIG, require its constraints to be in a normalized representation, such as in *conjunctive normal form (CNF)* [3]. Transforming a feature model into a CNF is always possible, as all its constraints consist of propositional formulas. To this end, for the remainder of the paper, we assume that the propositional formula that expresses the constraints of a feature model is in CNF. This means that each element c in the set of constraints C is a clause of a CNF. A clause then consists of a disjunction of arbitrarily many literals, which represent selecting features (i.e., positive literals) and deselecting features (i.e., negated literals). For instance, the constraint between the features in the file system OR-group can be expresses as the clause $\neg FS \vee EXT4 \vee NTFS \vee APFS$.

Configurations. A configuration is used to derive a variant of an SPL. Each configuration $config = (S, D)$ consists of a set of selected features S and a set of deselected features D, which are disjoint subsets of the feature model's feature set F. A feature is considered *selected* if it is contained in S, and *deselected* if it is contained in D. A feature that is neither in S or D is *undefined* in the configuration. If a configuration contains all features of a feature model (i.e., $S \cup D = F$) the configuration is *complete* and, otherwise, *partial*. A configuration is *valid* if all constraints of the respective feature model are satisfied by its selected and deselected features. For instance in Figure 1, a valid partial configuration is $config_1 = (\{Server, OS, Linux, FS, EXT4\}, \{Login\})$. In contrast, the partial configuration $config_2 = (\{Server, OS, Mac, FS, NTFS\}, \{Log\})$ is *invalid*, as it violates the cross-tree constraint $Mac \rightarrow APFS$.

Feature Model Anomalies. Due to a non-optimal set of constraints, a feature model may contain certain *anomalies*. Some of these anomalies are of interest when creating a MIG, namely *void* feature models, *core* and *dead* features as well as *redundant clauses*. A feature model is called *void* if no valid configuration exists for it. A feature is called *core* if no valid configuration exists in which the feature is deselected. Analogously, a feature is called *dead* if there is no valid configuration in which the feature is selected. If a feature is neither core nor dead, we call it a *configurable* feature. Figure 1 contains both, core and dead features. `Server` and `OS` can never be deselected, and thus are core. In contrast, `Login` is dead, as the constraint $Server \rightarrow \neg Login$ ensures that it can never be selected.

We consider two types of clause redundancy, *internal* and *external*. A clause is called *externally redundant* if it can be removed from the feature model without changing the spanned configuration space. Similarly, a clause is called *internally redundant* if it is possible to replace it with a clause that subsumes the original clause (i.e.,

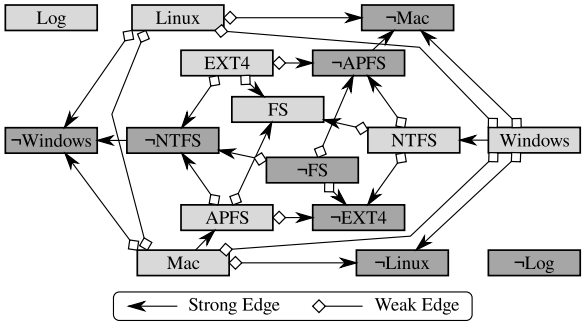

Figure 2: MIG for the `Server` feature model

contains fewer literals) without changing the configuration space. In our example in Figure 1, the clause $\neg Server \vee \neg Login$ is internally redundant, since $\neg Server$ always evaluates to *false*, and thus the clause is equivalent to the shorter clause $\neg Login$. The clause $Linux \vee NTFS \vee APFS$ is externally redundant as it can be derived from the ALTERNATIVE-group and other cross-tree constraints.

Feature Model Evolution. For the incremental build process, we consider two versions of a feature model such that an *evolution step* leads from the first to the second version. We define an evolution step as some change $\Delta = (F^+, F^-, C^+, C^-)$ applied to a feature model $M = (F, C)$ resulting in a modified feature model $M' = (F', C')$. The modified feature set F' results from removing the set of features F^- from F and then adding the set of features F^+ (i.e., $F' = (F \setminus F^-) \cup F^+$). Analogously, the modified set of constraints C' results from removing the set of constraints C^- from C and then adding C^+ (i.e., $C' = (C \setminus C^-) \cup C^+$).

2.2 Modal Implication Graphs

A *Modal Implication Graph (MIG)* is an extended implication graph that represents the dependencies of a feature within a feature model [17]. A MIG $G = (V, E_S, E_W)$ consists of a set of vertices $V = \{v_1, ..., v_n\}$, a set of strong edges $E_S = \{s_1, ..., s_m\}$, and a set of weak edges $E_W = \{w_1, ..., w_m\}$. Each vertex represents a single literal (i.e., the selection state of a feature) of one feature. Thus, the set of vertices of a MIG contains two vertices for each feature (i.e., selected or deselected). A *strong edge* $s = (v_a, v_b)$ is a directed edge that connects two vertices if the literal of v_a implies the literal of v_b. A *weak edge* $w = (v_a, v_b)$ is an edge that connects two vertices if the conjunction of the literal of v_a and at least one other literal implies the literal of v_b. In Figure 2, we depict the MIG for the Server feature model in Figure 1. For each node in the graph, we can see what a (de)selection of the corresponding feature implies for the selection state of other features. For instance, $\neg APFS$ has an outgoing strong edge to $\neg Mac$, because deselecting `APFS` implies deselecting `Mac`. Further, it has two outgoing weak edges to $EXT4$ and $NTFS$ as one of the two features `EXT4` or `NTFS` has to be selected when deselecting `APFS`. In contrast, Log and $\neg Log$ have no outgoing or incoming edges as the (de)selection of `Log` is not dependent on any other features.

Originally, MIGs were designed for an interactive configuration process to facilitate decision propagation (i.e., determining implied and excluded features) after (de)selecting a feature [17]. When (de)selecting a feature, the MIG can be traversed starting from the vertex that represents the feature in question and its new selection state (i.e., selected or deselected). All vertices that can be reached via a path that consists solely of strong edges (*strong path*) can be immediately (de)selected. Vertices that can be reached only via a path that contains at least one weak edge (*weak path*) must be investigated further using other reasoning techniques (e.g., a satisfiability (SAT) solver). All other vertices, which cannot be reached at all, can be ignored.

2.3 Building Modal Implication Graphs

One caveat of employing an additional data structure, such as a MIG, is that it has to be updated after evolution of the feature model. The naïve solution to achieve this is to simply rebuild the data structure using the new version of the feature model. The original build process of a MIG consists of three major steps, *analyzing the feature model*, *deriving nodes and edges*, and *optimizing the graph* [17].

Analyzing the Feature Model. As a first step, the given feature model is analyzed for certain anomalies with the goal of simplifying the upcoming graph structure. In particular, there are four possible anomalies that need to be identified, a *void feature model*, *core and dead features*, *externally redundant clauses*, and *internally redundant clauses*. By taking these anomalies into account, the number of features and clauses that are used to build the MIG can be reduced. A void feature model does not have any valid configurations so that it is not reasonable to build a MIG at all. As core and dead features can have only one selection state in all configurations, we do not need to create vertices for them, but only for configurable features. Both internal and external redundancies in constraints may artificially increase the number of weak edges in the graph, which can decrease its effectiveness. Thus, removing these redundancies bears potential to reduce the number of weak edges.

Deriving Nodes and Edges. The second step is to derive the initial graph structure. First, the set of vertices is created from the set of configurable features, which were determined in the previous step. For each configurable feature, two vertices are added to the graph, representing the feature's two possible selection states. Second, the set of strong edges is derived from the set of clauses that contain exactly two literals. Such a clause $a \vee b$ can be represented as two equivalent implications: $a \vee b \equiv \neg a \rightarrow b \equiv \neg b \rightarrow a$. Thus, two strong edges $s_1 = (\neg a, b)$ and $s_2 = (\neg b, a)$ are added to the graph. For example, the clause $\neg Windows \vee NTFS$ from Figure 1 results in the two edges $(Windows, NTFS)$ and $(\neg NTFS, \neg Windows)$ (cf. Figure 2). Finally, the set of weak edges is derived from the set of clauses containing more than two literals. Every clause $l_1 \vee ... \vee l_n$ can be written as $\neg l_i \rightarrow \bigvee_{1 \leq j \leq n, j \neq i} l_j$ for each literal l_i in the clause. Thus, for each clause $l_1 \vee ... \vee l_n$, two weak edges are added for each pair of literals within the clause such that $w_{i,j} = (\neg l_i, l_j)$ and $w_{j,i} = (\neg l_j, l_i)$ for all $1 \leq i < j \leq n$. For instance, for the clause $Linux \vee Windows \vee Mac$, the following six weak edges are added: $w_{1,2} = (\neg Linux, Windows)$, $w_{2,1} = (\neg Windows, Linux)$, $w_{1,3} = (\neg Linux, Mac)$, $w_{3,1} = (\neg Mac, Linux)$, $w_{2,3} = (\neg Windows, Mac)$, and $w_{3,2} = (\neg Mac, Windows)$ (cf. Figure 2). Note that clauses that contain none or only one literal can be ignored as these directly result in a void feature model or core/dead features, respectively.

Optimizing the Graph. Finally, the initial graph is optimized by adding and removing edges with the goal of maximizing the number of strong edges and minimizing the number of weak edges. A MIG can be considered more effective the less weak edges it contains, as this also lowers the number of weak paths between nodes. The number of weak edges is reduced by checking whether it is possible to convert weak paths to strong edges. For instance, in Figure 2 the MIG is not optimized. There is a weak path from $Linux$ to $EXT4$ (e.g., via $\neg Windows$, Mac, $APFS$, and $\neg NTFS$). Although there is no direct strong edge between both nodes the statement $Linux \rightarrow EXT4$ is always true due to the other constraints, and thus we can add an implicit strong edge from $Linux$ to $EXT4$.

Furthermore, the hull of all strong edges is built to facilitate traversing strong paths. For each vertex in the graph, a breadth-first search finds all other vertices that are reachable via a strong path. If there exists such a path, but no direct strong edge from the start to the end vertex (i.e., a transitive strong edge), this edge is added to the graph.

3 TIME-CONSUMING OPERATIONS IN THE BUILD PROCESS

The original build process for a MIG can be a time-consuming task for a large and frequently evolving feature model. However, only certain steps of the build process are computationally demanding. For the incremental build process, we investigate which operations in the original build process are the most time-consuming. We then focus on these steps to reason about whether it is possible to improve their efficiency either directly or with the help of the additional information provided in an incremental build process. We show the details of how we improve these operations in Section 4.

3.1 Identifying Time-Consuming Operations

In a preliminary evaluation, we built multiple MIGs on different feature models and measured the execution times of all distinct operations within the original build. As a result, we identified five particularly time-consuming operations: the *transformation* of the feature model's constraints *into CNF*, the building of the *strong hull* as well as the detection of *core and dead features*, *external clause redundancies*, and *implicit strong edges*. All other operations, such as the detection of internal clause redundancies and deriving the graph structure, are computationally undemanding and can be executed in negligible time compared to the other operations.

Almost all operations that use a satisfiability (SAT) solver (i.e., core/dead features, external clause redundancies, and implicit strong edges) are among the time-consuming operations, as these analyses solve NP-complete problems. The exception is the detection of void feature models, as this analysis requires only a single SAT query, which is relatively fast for most feature models [22]. In theory, building the strong hull of a MIG is an efficient operation that runs in polynomial time. However, in practice, our measurements showed that it required a significant part of the building time.

The transformation of a feature model's constraints into CNF is also a hard problem in general. For many feature models, this is a fast operation, but feature models with complex constraints exist that are hard to transform into a CNF (e.g., Linux [30]). However, this is only of concern if we consider the MIG build process in isolation. As a matter of fact, the CNF representation is needed for almost all common feature model analyses. Thus, either it was already built for another analysis or will probably be reused by other analyses after building a MIG. For this reason and because it is technically not part of the build process, we consider this operation to be out of scope for this paper.

3.2 Building the Strong Hull

Building the strong hull of a MIG is a graph traversal problem. Therefore, we were surprised of its rather long execution time. A further investigation showed that the reason for this inefficient operation was an issue in the original implementation [17]. By changing the implementation, such that it works on an adjacency list instead of an sparse adjacency matrix, we were able to substantially speed-up this operation. We now use this improvement in both, the original and the incremental build process.

3.3 Operations Using a SAT Solver

All operations in the build process that employ a SAT solver are similar in nature, as they all analyze the feature model with multiple similar SAT queries. This allows us to reason about potential efficiency improvements of these three operations together. For all operations employing a SAT solver, we see three possible improvements: generally *improving the performance*, *reducing* the amount of *SAT queries*, and entirely *skipping the operation*.

General Performance Improvement. One possible improvement is to generally speed up the operation by using more efficient analyses techniques or possibly other solvers. This would benefit both, the original and the incremental build process. However, the existing analyses using SAT solvers in these operations are already fairly optimized [2, 21]. Furthermore, in this paper, we focus on enabling an incremental build, rather than general improvements. Thus, generally improving these operations is out of scope for this paper.

Reducing SAT Queries. Another way to speed up the operations is by reducing the number of overall SAT queries as these are the most expensive parts. Each SAT query in every operation can be mapped to exactly one particular anomaly (i.e., a core feature, dead feature, redundant clause, or implicit strong edge). If we knew or could estimate these anomalies in advance (e.g., whether a feature is core or a clause redundant), we could avoid the respective SAT queries. Regarding the additional information from the incremental build, we can think of two ways of achieving this. First, we can reason about whether and how the set of anomalies for the previous feature model has changed. Second, given the feature model change, we can use heuristics to estimate whether a new anomaly will occur.

Given a feature model and corresponding MIG of the previous version, we can derive its core and dead features, redundant clauses, and implicit strong edges (cf. Section 4.1). Then, by analyzing the feature model change, we can determine whether these anomalies will change. Adding constraints to the feature model can add more anomalies, but does not remove existing anomalies. This includes new core and dead features, redundant clauses, and implicit strong edges. In contrast, removing constraints may only remove existing anomalies, but never adds new ones.

Instead of investigating all features and constraints for anomalies, by using a heuristic, we can test only the ones that are most likely to be affected by a change (e.g., by considering only features that appear in added or removed constraints). Testing only a subset of features and constraints decreases the number of SAT queries. However, using heuristics introduces the risk of decreasing completeness of a computed MIG, i.e., that the graph contains fewer strong edges than possible and more weak edges than necessary. In turn, this may decrease the MIGs effectiveness in terms of facilitating decision propagation.

Skipping Operations. Lastly, we may choose to skip an entire operation during the build process. Similar to using heuristics, this can also reduce the completeness of the MIG. Based on the usage scenario, it may or may not be sensible to skip operations. For instance, if a MIG (version) is created only once and it is used for many thousands of configuration processes, the additional effort to not skip the operations may pay off in the long term. In contrast, if a MIG (version) is only used for few configuration processes, it pays off to skip them. This method can of course be applied to the original and incremental build process. When first introducing MIGs [17], we already considered this possibility for the operation of detecting implicit strong edges, as this is by far the most time-consuming operation. Therefore the entire operation was already optional in the original build process and skipping it results in an *incomplete MIG*, which is than less effective during decision propagation.

4 INCREMENTAL MODAL IMPLICATION GRAPHS

To apply an incremental build process instead of the original one, there must be a MIG built for a feature model and a new version of this feature model (e.g., stemming from evolution). The incremental build process is based on the original build process and follows the same three major steps, but uses the modified variants of the most time-consuming operations. The core idea of the incremental build is to reuse information on anomalies from an older version of the graph and the information on the feature model change to reduce the number of SAT queries. As the creation of the graph structure (i.e., adding vertices and edges) is an efficient operation, we rebuild the graph from scratch. This simplifies the implementation compared to an implementation that needs to be able to add and remove edges and vertices to and from a graph.

In the following, we describe the key improvements of our new incremental build process over the original build process. To this end, we explain how we determine the change to a feature model and how the type of change affects the build process. Further, we focus on the adaptions to the three time-consuming operations, the detection of *core and dead features*, *external clause redundancies*, and *implicit strong edges*.

4.1 Computing the Feature Model Change

For the incremental build process, we first infer the change Δ between the two feature models M (i.e., before the evolution) and M' (i.e., the current version). First, we compute the unified feature set between both feature model versions by adding F and F' (i.e., $F_U = F \cup F'$). Second, we adapt all clauses in C and C' such that they use the same variable set F_U. This is a necessary step to ensure that a renamed feature is not treated as two separate variables in C and C', but the same variable. Third, we infer the set of removed clauses and added clauses by computing $C^- = C \setminus C'$ and $C^+ = C' \setminus C$.

From Δ, we can reason about the type of change that is necessary to update the MIG. In particular, we characterize Δ depending on the change to the set of constraints. Δ can either *do no change* (i.e., $C^- = C^+ = \emptyset$), *add constraints* (i.e., $C^- = \emptyset$ and $C^+ \neq \emptyset$), *remove constraints* (i.e., $C^- \neq \emptyset$ and $C^+ = \emptyset$), or *replace constraints* (i.e., $C^- \neq \emptyset$ and $C^+ \neq \emptyset$). We do not consider the change to the feature set F (i.e., if $F \neq F'$), as changing this set does not change the relationships between features. Naturally, adding or removing features to a specific feature model representation, such as a feature diagram, also changes related constraints. However this is then reflected in the set of constraints C. When the set of features F changes, it suffices to update the MIG by adding and removing the respective nodes, which is part of the second building step *deriving nodes and edges*. Adding constraints to C may cause new anomalies within a feature model, but will not remove existing ones. Removing constraints from C may fix old anomalies, but will never cause new ones. Replacing constraints can be seen as simultaneous addition and removal of constraints to a feature model, and thus may introduce *and* fix anomalies.

In addition to the feature model change Δ, we also require information on anomalies of the previous feature model M. We determine the set of anomalies from the CNF and MIG of M. Core and dead feature, as well as implicit strong edges, are saved within a MIG, and, thus, we can access them without further computation. We derive the set of previously externally redundant clauses by comparing the set of clauses in the CNF with the edges in the MIG. If there is a clause with no corresponding edge, it was redundant.

4.2 Detecting Core and Dead Features

For detecting core and dead features, we use an analysis based on querying a SAT solver. For each feature, the analysis assumes that the feature is selected and then uses the SAT solver to find a valid configuration under this assumption. If the SAT solver cannot find a valid configuration, the feature must be dead. Analogously, if there is no valid configuration when the feature is assumed to be deselected, the feature is core.

Even though this analysis requires SAT queries, it is orders of magnitudes faster compared to the following two operations. This is due to the re-use of already found SAT solutions to substantially reduce the number of SAT queries [21]. For this matter, we make only small adaptions to this operation for the incremental build process. We split the operation in two parts: First, we check whether the core/dead features from the previous MIG are still core/dead. Second, we check whether any previously configurable features are now core or dead. This allows us to avoid unnecessary SAT queries, if constraints were only added or only removed. We execute the first part only if Δ is removing or replacing constraints and execute the second part only if Δ is adding or replacing constraints. If Δ makes no change to the constraints at all, we skip both parts.

We do not consider using any heuristics for this operation or skipping it entirely, for two reasons. First, the operation is by far the fastest compared with the other operations that employ a SAT solver. Second, an incorrect result from this operation could severely

harm the graph structure as it could add vertices that cannot be part of a valid configuration (i.e., false negative) or neglect vertices that are necessary (i.e., false positive).

4.3 Detecting External Clause Redundancies

The operation starts with an empty formula and incrementally adds clauses from the feature model one at a time. For each clause, it checks whether the conjunction of the current formula and the complement of the clause is a contradiction (i.e., not satisfiable). If so, the clause is implied by the current formula and therefore redundant. In this case, the operation removes the clause from the set of constraints. Otherwise, the clause is added to the formula.

Similar to the detection of core and dead features, we split the operation in two parts, checking whether the old externally redundant clauses are still redundant and checking whether there are new externally redundant clauses. If and only if Δ removes or replaces constraints, it is necessary to check whether old redundant clauses are still redundant. Checking redundancy of previously redundant clauses is mandatory. If a former redundant clause is no longer redundant but is not added to the MIG, the resulting MIG would be incorrect.

Checking whether previously non-redundant clauses have become redundant is optional, as it does not impact the correctness of the MIG, but only increases its number of edges. Therefore, we consider three different options if Δ adds or replaces constraints. First, checking all previously non-redundant clauses for redundancy. Second, using a heuristic to test only a subset of the previously non-redundant clauses. In this case, a clause is checked for redundancy only if it contains at least one feature from any clause in the set of added constraints C^+. Third, skipping the second part of the operation and do not check any previously non-redundant clauses. Clearly, only the first option guarantees completeness for finding all externally redundant clauses. However, it also does not have any performance improvement compared to the original build process. Thus, we favor the second and third option, which both may introduce redundancy to the new MIG. In our evaluation, we test performance and potential loss of completeness of both options.

4.4 Detect Implicit Strong Edges

For each vertex in the graph, the operation uses a breadth-first search to find all other vertices that are reachable via a weak path. If there exists such a path, but no strong path from the start to the end vertex, the operation tests whether the literal of the start vertex implies the literal of the end vertex. For this, the operation uses the SAT solver to test whether the conjunction of the feature model formula and the complement of the implication is a contradiction. In this case, the operation adds a strong edge and removes the corresponding weak edges.

For this operation, we make very similar adaptions as for the operation of detecting external clause redundancies for the same reasons. We split the operation in two parts, checking if previously implicit strong edges are still valid and checking if we can derive new implicit strong edges. If Δ removes or replaces constraints, we must check whether every previously implicit strong edge is still valid. If Δ adds or replaces constraints, we again consider the three options of running the entire second part of the operation, using a heuristic to investigate only a subset of weak edges, or skip the second part of the operation altogether. For the second option, we use a similar heuristic as for detecting redundancies. In particular, we only test weak edges that contain at least one feature from any clause in C^+. Again, the first option does not have any performance benefit, but is the only one that leads to completeness of the resulting MIG. Thus, we evaluate the performance and loss in completeness of the second and third option in our evaluation.

5 EVALUATION

The build time of a MIG is a limiting factor on its usefulness for frequently evolving feature models. To evaluate whether our concept for incrementally building MIGs is able to overcome this limitation, we compare it to the original build process. In particular, we pose three research questions that enable us to assess the incremental build process. First, we examine whether the incremental build process improves the MIG build time compared to the original build process. Second, as the incremental build process uses some heuristic analysis, we are also interested in whether usage of an incrementally built MIG is less efficient than an original MIG. Third, considering this potential trade-off between build time and completeness, we investigate under which circumstances it is suitable to use the incremental instead of the original build process. In particular, we want to know whether there is an indicator within the change information that lets us reason about the suitability of the incremental build process. In summary, we aim to answer the following research questions:

- RQ_1 Does an incremental MIG build process improve performance compared to an original build process?
- RQ_2 How much does the loss in completeness of an incrementally built MIG impact its effectiveness in analyses?
- RQ_3 Can the usefulness of an incremental build process be inferred by the characteristics of a feature model change?

In our evaluation, we use the evolution histories of four real-world systems to build a MIG for many different feature-model versions with the original and the incremental build process. Then, we use the resulting MIGs in decision propagation analysis to examine their performance.

5.1 Evaluation Setup

First, we explain the design of our experiments in the evaluation. We describe which are the relevant variables we measure in order to answer our research questions. Second, we present our subject systems and their respective evolution histories. Third, we explain all additional parameters in our experiments. Each experiment uses a subject system, a pair of versions from the corresponding evolution history, specific parameter settings for the original and incremental build processes, and a particular MIG as input for the incremental build. We vary these parameters to get a more comprehensible understanding of the differences between the original and incremental build process. Lastly, we provide relevant details on our implementation of the incremental build process and the evaluation environment.

5.1.1 Measurements. In our experiments we measure two different variables: the time to build a MIG and the time of a decision propagation using the built MIG. In the first part of each experiment, we build an original and an incremental MIG for a given feature model version of a subject system. We separately measure how much time it takes for both build processes to finish. As mentioned in Section 3, the CNF transformation of a feature model is not part of the build process, and thus is also not measured in the experiments.

In the second part of each experiment, after building both MIGs, we use each MIG in a series of decision propagation analyses to measure if there is any difference in their performance. To this end, we choose 200 random literals from the feature model and perform a decision propagation with each of the chosen literals as starting point (i.e., (de)selecting it in a configuration). We measure the time it takes for each decision propagation to finish and sum up the results for each MIG. We do not allow duplicates in the random literal list and use the same literals for every MIG on the same version of a feature model.

5.1.2 Subject Systems. We use the version histories of four real-world systems. For each system, we have one feature model per version in the system's history. In our repository[2], we provide access to almost all[3] of the feature models versions. *Busybox* (500 – 700 features) and *Linux* (16,000 features) are software systems from the operating system domain. The evolution history of *Linux* contains 14 versions, ranges from November 2013 to December 2013, and has short and long times between versions (i.e., within a day and within a month). In case of *Busybox*, we have two version histories from May 2007 to May 2010 for the same system that differ in the time between versions. For *Busybox (Commits)*, each of its 187 versions corresponds to the state of the system after a commit in its version control system that changed the feature model. In contrast, the history of *Busybox (Monthly)* contains 37 monthly snapshots of the system. *FinacialServices01* (500 – 700 features) comes from the financial domain and represents a family of financial products rather than software. Its evolution history spans from May 2017 to March 2019 and contains 20 monthly snapshots. *Automotive02* (14,000 – 19,000 features) is cyber-physical system from the automotive domain. Its evolution history contains 6 monthly snapshots.

5.1.3 Evolution History. In order to perform an incremental build, at least one MIG from a previous version is required. For each of our subject systems, we have an evolution history that contains multiple consecutive versions (i.e., $\overrightarrow{H} = \{v_1, ..., v_n\}$). For any pair of versions (v_a, v_b), we can use the MIG of the first version v_a to incrementally build the MIG for the second one v_b. Note that it is not necessary to use the MIG from the directly preceding version, but any preceding version.

To see the impact of different evolution steps, we test different scenarios that lead to a different list of version pairs. The most natural approach would be to evaluate the incremental build process with regard to all consecutive feature model versions (i.e., v_1 and v_2, v_2 and v_3, and so on). However, this would require that an original MIG is available for each version, which is then used to incrementally build the MIG for the next version. Thus, we also consider the scenario that there is only one original MIG from the first version of a model. In particular, we consider three different lists of version pairs derived for any given evolution history, namely *consecutive*, *accumulative*, *sequential*.

Consecutive. With the consecutive version pair list, we aim to show the impact of building an incremental MIG for every new version based on the MIG of the directly preceding version. For a given evolution history, we construct version pairs, such that each version is combined with its direct predecessor (i.e., $V = \{(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)\}$). For incrementally building a MIG for any version, we use the original MIG from the preceding version as input.

Accumulative. With the accumulative version pair list, we aim to demonstrate the impact of only using an incremental build process over several consecutive versions. We use the same version pairs as for the consecutive version pair list (i.e., $V = \{(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)\}$). However, instead of using the original MIG from the preceding version, we use an incrementally built MIG as input. Only for the first pair (v_1, v_2), we use the original MIG of v_1.

Sequential. With the sequential version pair list, we aim to show the impact of skipping some versions. For a given evolution history, we construct version pairs, such that each version is combined only with the initial version (i.e., $V = \{(v_1, v_2), (v_1, v_3), ..., (v_1, v_n)\}$).

5.1.4 Parameters. As stated in Section 4, we can choose different options for the two most time-consuming operations of detecting *external clause redundancies* and *implicit strong edges*. These options influence the execution time of the build process and the completeness of the resulting MIG. For the original build process, we have two options for each operation, either performing it (*yes*) or skipping it entirely (*no*). For the incremental build process, the options depend on the original build process. If we skip an operation in the original build process (*no*), we also have to skip this operation in the incremental build as well (*no*). This is due to the fact that we use the result of the operations from the MIG of an earlier version and if an operation was skipped the required information is missing. If we performed the operation in the original build process (*yes*), we have two options, using a heuristic to speed up the second part of the operation (*heuristic*) or skipping the second part of the operation entirely (*skip*). Combining the different values for these options, results in the following five parameter settings that we use for our experiments:

	Original		**Incremental**	
ID	*Redundancy*	*Strong*	*Redundancy*	*Strong*
1	*no*	*no*	*no*	*no*
2	*yes*	*no*	*skip*	*no*
3	*yes*	*no*	*heuristic*	*no*
4	*yes*	*yes*	*heuristic*	*heuristic*
5	*yes*	*yes*	*skip*	*skip*

5.1.5 Implementation. We base our implementation of the incremental build process on the implementation of the original build process, which is part of the FeatureIDE framework [15, 20][4]. The implementation is written in Java and uses Sat4J [18] as a SAT

[2] https://github.com/skrieter/evaluation-mig

[3] We omit 10 versions of FinacialServices01 due to reasons of confidentiality.

[4] https://featureide.github.io/

Table 1: Absolute and relative build and usage times for all systems and parameter settings.

System	Parameters	Build Time					Usage Time				
		Orig (s)	Inc (s)	Ratio			Orig (s)	Inc (s)	Ratio		
		∅	∅	*Min*	∅	*Max*	∅	∅	*Min*	∅	*Max*
Busybox (Commits)	1	0.002	0.003	0.478	0.823	1.690	0.034	0.034	0.771	0.997	1.052
	2	0.012	0.004	0.311	3.176	6.333	0.034	0.034	0.662	0.997	1.050
	3	0.012	0.006	0.275	2.416	5.266	0.034	0.034	0.959	1.002	1.070
	4	0.086	0.018	0.283	12.732	32.996	0.034	0.034	0.955	1.000	1.204
	5	0.086	0.004	0.301	21.094	47.502	0.034	0.034	0.948	1.001	1.191
Busybox (Monthly)	1	0.003	0.003	0.589	0.766	0.919	0.040	0.040	0.973	0.999	1.026
	2	0.013	0.004	1.998	2.981	4.226	0.040	0.040	0.972	0.999	1.031
	3	0.013	0.008	0.911	1.990	3.490	0.040	0.040	0.970	0.997	1.021
	4	0.102	0.025	1.296	8.614	26.702	0.041	0.041	0.966	1.014	1.058
	5	0.102	0.005	10.534	19.867	37.456	0.041	0.040	0.986	1.034	1.056
FinacialServices01	1	0.161	0.165	0.925	0.977	1.031	0.198	0.197	0.929	1.006	1.069
	2	0.345	0.182	1.336	1.913	2.348	0.195	0.197	0.934	0.988	1.030
	3	0.341	0.333	0.841	1.038	1.392	0.196	0.197	0.956	0.995	1.049
	4	18.809	13.218	0.969	1.604	2.992	0.195	0.193	0.963	1.009	1.060
	5	18.837	6.792	0.853	9.975	23.033	0.195	0.192	0.974	1.017	1.051
Automotive02	1	3.109	3.535	0.822	0.882	0.958	7.490	7.502	0.992	0.999	1.005
	2	8.583	3.873	1.672	2.120	3.395	7.451	7.499	0.985	0.994	0.999
	3	8.561	9.120	0.690	1.080	1.616	7.447	7.458	0.993	0.998	1.005
	4	2090.844	1547.998	1.069	4.244	13.733	7.353	7.365	0.992	0.998	1.005
	5	2095.159	4.270	110.037	424.784	1221.410	7.362	7.471	0.977	0.986	1.001
Linux	1	29.326	29.446	0.829	0.998	1.127	15.686	15.677	0.992	1.001	1.010
	2	2218.864	132.196	15.514	16.806	18.269	15.667	15.664	0.993	1.000	1.004
	3	2228.968	2845.951	0.748	0.783	0.815	15.671	15.665	0.993	1.000	1.008

solver. We modify the current source code in FeatureIDE by adding some general improvements, which result in the implementation of the original build process that we use in our experiments. This includes fixing the operation of building the strong hull, which we described in Section 3. We then use this modified implementation of the original build process as basis for the incremental build process and replace the operations discussed in Section 4 accordingly. Both build processes use the same underlying hardware and software framework (e.g., for loading feature models and using decision propagation with a MIG). In detail, we use the following hardware specifications: *CPU*: Intel Core i7-5500U (2.4 GHz), *Memory*: 16 GB, *OS*: Linux 5.10.23-1-MANJARO, *JVM*: OpenJDK 64 Bit 15.0.2.

5.2 Evaluation Results

In Table 1, we show an overview of our measurements for all subject systems (cf. Section 5.1.2) and all parameter settings (cf. Section 5.1.4). All values in the table are aggregated over all version pairs (cf. Section 5.1.3). We show the mean time in seconds for building an original and incremental MIGs and the mean time in seconds for executing decision propagation with the original and the incremental MIGs. In addition, we show the ratio between original and incremental MIG for the build and usage time. For the ratio, we state the min, mean, and max values, respectively.

In addition to Table 1, we show a more detailed plot of the build time ratio in Figure 3. The figure contains one scatter plot per parameter setting (columns) and version pair list (rows). Each plot contains the data of all measured values for the particular parameter setting and version pair list. The y-axis shows the ratio on a log scale (i.e., the higher the more time was saved by the incremental build). The x-axis shows the index of the used version divided by the number of versions in the evolution history of the system. This normalization is done in order to evenly space out all systems over the x-axis. Additionally, each plot contains a regression curve per system to visualize any trend in the data.

From our data table and plots we can make three observations. First, the differences in the data for different version pair lists are relativity small. All plots in a column roughly show the same behavior with the exception of the sequential version pairs (third row) for parameter settings 3 and 4. We can see that the performance improvement of the incremental build process is better if the versions are closer together. For example, for parameter setting 4, *Busybox (Commits)* (v_1, v_2) has a speed-up of factor 10, in contrast to factor 2 for (v_{186}, v_{187}). Second, for parameter setting 1, we can see no benefit of using an incremental build process over the original one. There is even a small overhead (i.e., factor 0.76 – 0.99 on average) that increases the overall build time. This can be explained by the time it takes the incremental build process to compute the feature model change. Further, none of the two most time-consuming operations that were modified in the incremental build can be utilized as they are skipped for both build processes. Third, there is a moderate improvement for parameter settings 2 and 3 (i.e., factor 0.78 – 16.8 on average) and a high improvement for parameter settings 4 and 5 (i.e., factor 1.6 – 424.78 on average). This is expected as settings 2 and 3 enable the detection of externally redundant clauses and

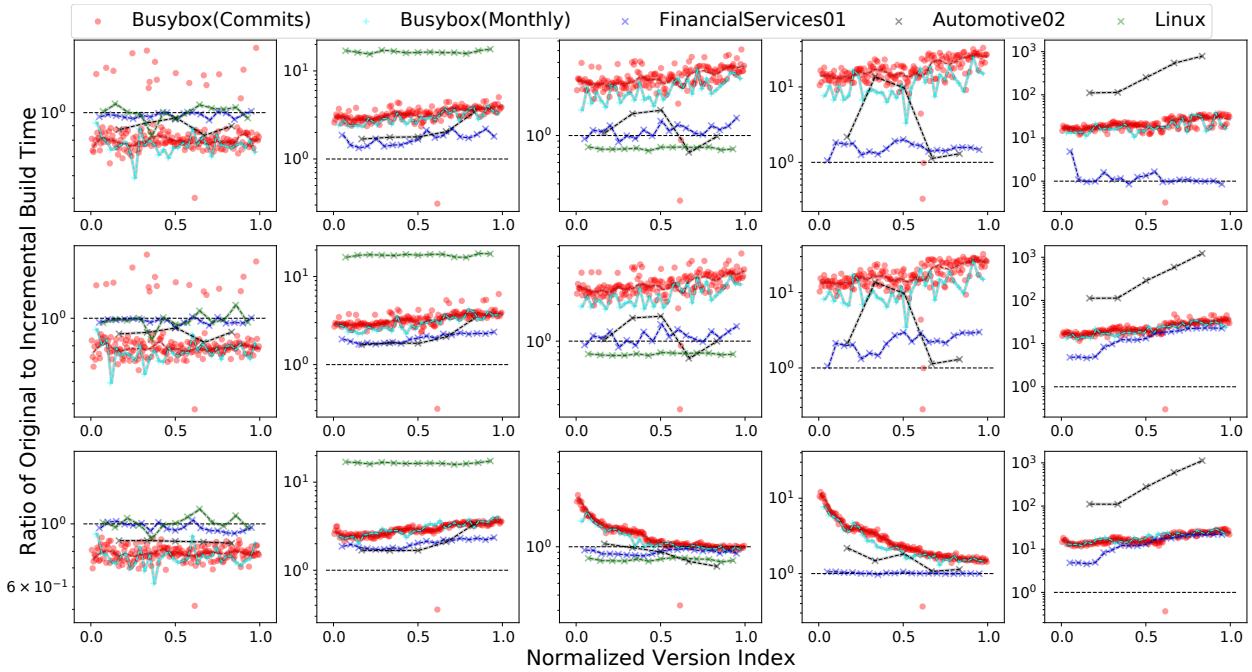

Figure 3: Build time ratio (original/incremental) for all systems and versions (X: Normalized versions difference; Y: Ratio; Columns: Parameters 1–5; Rows: Consecutive, Accumulative, Sequential)

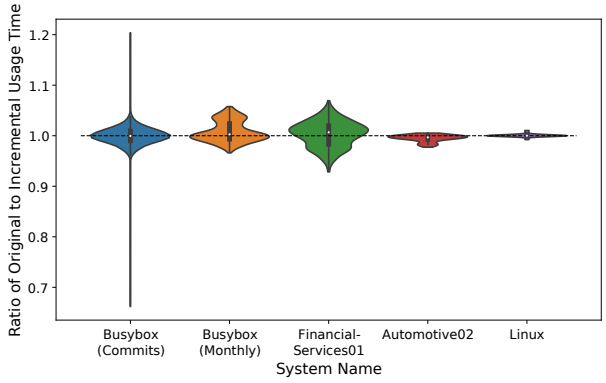

Figure 4: Aggregated usage time ratio (original/incremental) for all systems

settings 4 and 5 also enable the detection of implicit strong edges. Both operations are partially skipped during the incremental build process. There is also a clear difference between skipping the second part of the respective operations and using a heuristic. When using a heuristic, it substantially decreases the build time improvement of the incremental build compared to skipping(e.g., from factor 4.24 to factor 424.78 for *Automotive02*).

In Figure 4, we show the ratio of the usage time of an original and an incremental MIG aggregated for each system. The y-axis shows the ratio and the x-axis the system. We aggregated the data, as there is almost no difference in the data distribution for different parameter setting and version pair lists. These results lead us to the following observation. The difference in usage time for original and incremental MIGs is almost non-existent. Although, we can see some relative differences for *Busybox* and *FinancialServices01* however in absolute terms the difference is within a few milliseconds (e.g., the maximum time difference for *Automotive02* is 109 ms). Both the ratio and the actual difference are negligible for a single configuration process. The practical difference in the MIG's completeness also seems to be independent from the chosen parameter settings and version distance.

5.3 Discussion

RQ_1. The incremental build process is able to drastically outperform the original build. This is dependent on the concrete parameter settings, though. When using a light-weight original build process that does not detect redundancies or implicit strong edges, and thus computes an incomplete MIG, the incremental build process has no benefit. Only when the original build process aims for a complete MIG, the incremental build process is able to achieve substantially lower building times (varying from a factor of 10 to 400 on average). In these cases, it seems to be more efficient to not rely on a heuristic, but skipping entire parts of certain operations within the incremental build, as this always improves the performance (e.g., in case of *Automotive02* from factor 4 to factor 400 on average).

RQ_2. The incremental build process does barely affect the resulting MIGs effectiveness within decision propagation. An incremental MIG cannot be guaranteed to be complete regardless of the considered parameter settings. Thus, there is a theoretical loss in completeness for the incremental build process. However, the practical

performance impact when using an incremental MIG compared to a complete original MIG is almost not noticeable in our experiments (within a factor of 1.01 on average). This is in line with our findings from our first paper on MIGs, where we found that a complete MIG only slightly improves decision propagation compared to an incomplete MIG.

RQ_3. The difference in usage time between complete, incomplete, and incremental MIGs is relatively small. On the other hand, the build time varies dramatically for different parameter settings and whether an original or incremental build process is used. Thus, it makes sense to use the incremental build frequently to keep build times low and to use incrementally built MIGs as input for the next incremental build process. One could argue that it is more beneficial to only use incomplete MIGs all together (i.e., using the original build process with parameter setting 1) as this results in the fastest build process and only small loss in effectiveness. However, there are circumstances, where the incremental build process is still superior, for instance, when a complete MIG is already present (e.g., from other analyses). In this case, using an incremental build is more suitable than rebuilding an (in)complete MIG from scratch. Overall, for the incremental build process shows a substantial speed-up for all parameter settings (except 1) without noticeable loss in effectiveness, even for large evolution steps.

5.4 Threats to Validity

In the following, we reason about possible threats to validity within our evaluation and explain what we did to mitigate potential biases.

5.4.1 Internal Validity. There may be several causes for a computational bias. First, the JVM may influence the required time for consecutive runs due to just-in-time compilation. To tackle this issue, we performed warm-up computations prior to the first measurement. Second, Java regularly frees the memory of not referenced objects by means of garbage collection. To mitigate this effect, we instructed the JVM to run the garbage collector before building and using a MIG. Third, there may be a general computational bias, which can cause minor differences in measured execution times. To reduce the mitigate this bias, we performed three repetitions for each computation and used the median of those.

For the entire empirical evaluation, we use the CNF transformation implemented in FeatureIDE. As both, the original and incremental build process, rely on a CNF input, using a different CNF may change the internal structure of a MIG, and thus may influence measured execution times.

We randomly (de)selected features to evaluate decision propagation with different MIGs. This may result in random bias, where we by chance only picked features that result in certain corner cases of decision propagation. To mitigate this random bias, we (de)selected 200 different features and made sure that every MIG is tested against the same list of features.

5.4.2 External Validity. Our evaluation results maybe cannot be generalized for other evolution histories or other configurable systems. There are only very few histories of industrial systems publicly available and we limited ourselves to such systems (contrary to artificial ones) for more expressive results on real-world scalability. Nevertheless, we evaluated the history of four systems from very different domains that are widely used for empirical evaluations [25–27, 29].

6 RELATED WORK

We introduce an incremental build process for updating MIGs. However, the idea of incremental analysis is not novel. Especially incremental SAT solving [23] is related to our approach. Incremental SAT solving improves the performance for solving consecutive SAT queries, in which only small parts of each query are changed by reusing information from previous solutions [11–13, 24]. Incremental SAT solving could be used complement our approach in two ways. First, as we are are solving multiple similar SAT queries within the MIG build process, an incremental SAT solver may be able to speed-up some operations. This would then apply to the original and incremental build process. Second, incremental SAT solvers could be used in conjunction with the analysis result available from previous MIGs to facilitate the build process even more. If the feature model change is small all SAT analyses within the build process could be speed-up by providing them with SAT solutions from a previous feature model version.

MIGs are a type of supplementary data structure that represent configuration knowledge. There are other data structure, such as *Binary Decision Diagrams (BDDs)* [6, 10], with a similar purpose. BDDs could be used instead of MIGs in the task of decision propagation and also other analyses [14]. While BDDs may be more effective in facilitating certain analysis tasks, they are harder to build for large and complex feature models [31]. Therefore, MIGs can be seen as an alternative that is less effective but faster to create, and thus might be more suitable in certain situations.

7 CONCLUSION

Supporting data structure for configurable systems, such as *Modal Implication Graphs (MIGs)*, must be updated after every evolution of the corresponding feature models. In this paper, we introduced a concept for an incremental build process of MIGs in order to speed-up the creation of a MIG after feature model evolution, which enables developers to utilize the benefits of employing a MIG for frequently evolving feature models. We identified time-consuming operations in the original build process and suggested modifications to these operations that are able to improve the performance of the overall build process. The modifications are based on reusing information of a previously computed MIG and the feature model change with respect to the previous feature model version. Further, we use heuristics in the incremental build process, which lower the overall build time, but may result in a MIG with a non-optimal set of edges (i.e., incomplete), which may lower its effectiveness in later usage. In our evaluation, we found that in some scenarios, the incremental build process can outperform the original build process by orders of magnitude. Furthermore, there appears to be almost no difference in effectiveness using an original or an incremental MIG within decision propagation. We conclude that the incremental build process is a suitable method for updating a MIG after feature model evolution. In future work, we plan to explore further possibilities to improve the original and incremental build process, for instance by using different automated reasoning techniques or incorporating incremental SAT solving.

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[2] David Benavides. *On the Automated Analysis of Software Product Lines Using Feature Models - A Framework for Developing Automated Tool Support*. PhD thesis, University of Seville, 2007.

[3] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010.

[4] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 7:1–7:8. ACM, 2013.

[5] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*, pages 73–82. ACM, 2010.

[6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[7] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[8] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Trans. Software Engineering (TSE)*, 34(5):633–650, 2008.

[9] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, 2000.

[10] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002.

[11] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[12] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electron. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[13] Katalin Fazekas, Armin Biere, and Christoph Scholl. Incremental inprocessing in SAT solving. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2019.

[14] Rune M. Jensen. Clab: A C++ library for fast backtrack-free interactive product configuration. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, page 816. Springer, 2004.

[15] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. FeatureIDE: Empowering Third-Party Developers. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 42–45. ACM, 2017.

[16] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. YASA: yet another sampling algorithm. In *VaMoS '20: 14th International Working Conference on Variability Modelling of Software-Intensive Systems, Magdeburg Germany, February 5-7, 2020*, pages 4:1–4:10. ACM, 2020.

[17] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. Propagating Configuration Decisions with Modal Implication Graphs. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 898–909. ACM, 2018.

[18] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[19] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. Practical Pairwise Testing for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 227–235. ACM, 2013.

[20] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.

[21] Marcílio Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, 2009.

[22] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. SAT-Based Analysis of Feature Models is Easy. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 231–240. Software Engineering Institute, 2009.

[23] Malek Mouhoub and Samira Sadaoui. Solving incremental satisfiability. *Int. J. Artif. Intell. Tools*, 16(1):139–147, 2007.

[24] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Ultimately incremental SAT. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 206–218. Springer, 2014.

[25] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. Anomaly Analyses for Feature-Model Evolution. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*, pages 188–201. ACM, 2018.

[26] Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. Stability of product-line samplingin continuous integration. In *VaMoS'21: 15th International Working Conference on Variability Modelling of Software-Intensive Systems, Virtual Event / Krems, Austria, February 9-11, 2021*, pages 18:1–18:9. ACM, 2021.

[27] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. Product Sampling for Product Lines: The Scalability Challenge. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2019. To appear.

[28] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[29] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 667–678. ACM, 2016.

[30] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. Europ. Conf. on Computer Systems (EuroSys)*, pages 47–60. ACM, 2011.

[31] Thomas Thüm. A BDD for linux?: the knowledge compilation challenge for variability. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, pages 16:1–16:6. ACM, 2020.

[32] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45, 2014.