Fast Approximate Calculation of Valid Domains in a Satisfiability-based Product Configurator

Johannes Werner[1] and **Tomáš Balyo**[2] and **Markus Iser**[3] and **Michael Klein**[4]

Abstract. Calculating valid domains is an important feature of an interactive product configurator. Since it is an NP hard problem, it is necessary (for large real-world instances) to calculate valid domains only approximately in order to keep the response time low. In this paper, we present a new fast and accurate approximation algorithm to calculate the valid domains in a satisfiability based interactive product configurator. The algorithm is based on building a full implication graph during unit propagation and performing a search in that implication graph in order to approximate whether a domain value is valid. We experimentally compared our new algorithm to the algorithm used by the commercial SAT-based configurator CAS Merlin and measured speedups of up to 18-fold while maintaining the same accuracy.

1 Introduction

With the ever increasing number of products available on the market, customers increasingly demand products that are tailored to their specific needs. The goal of mass customization is to deliver products and services that best meet individual customers needs with near mass production efficiency[16].

In this paper, we focus on configuration systems that offer an *interactive configuration process*. In such systems, the configurator provides feedback on the current configuration after every new user decision [9]. In a satisfiability (SAT) based configurator the constraints of a product are represented as a Boolean formula and checking whether a user decision is valid under the constraints and previous user decisions amounts to solving a SAT problem. To better guide the configuration process a configurator should also provide feedback about the remaining valid options for future decisions. This can also be achieved by a series of SAT solver calls [9]. However, this can lead to long calculation times, therefore we will over-approximate valid options instead of calculating them precisely.

The main contribution of this paper is a new algorithm that is able to approximate valid domains in the context of interactive product configuration. We experimentally compared our new algorithm to the algorithm used in the commercial configurator CAS Merlin [2] and measured that our algorithm runs up to 18 times faster while approximating the valid domains with the same accuracy. We further experimentally analyzed how accurate the approximation is compared to the complete set of valid domains. We found that the approximation is identical to the exact solution in most cases, we observed a difference only in 4% of the test instances.

[1] CAS Software AG, johannes.werner@cas.de
[2] CAS Software AG, tomas.balyo@cas.de
[3] Karlsruhe Institute of Technology, markus.iser@kit.edu
[4] CAS Software AG, michael.klein@cas.de

2 Preliminaries

2.0.1 *Propositional Satisfiability*

A *propositional variable* can be either assigned the value $true$ or $false$. A *literal* is a propositional variable (e.g. x_1) or its negation (e.g. $\neg x_1$). A disjunction (or, $\vee$) of literals is called a *clause* (e.g. $(x_1 \vee \neg x_2 \vee \ldots)$) and a conjunction (and, $\wedge$) of clauses is called a conjunctive normal form (CNF) formula, or just formula (e.g. $(x_1 \vee \neg x_2) \wedge (\neg x_3) \wedge \ldots$).

The *satisfiability problem* (SAT) is the task to decide whether a given formula F has a variable assignment that evaluates F to $true$ (i.e. satisfies F, or F is satisfiable). An assignment satisfies F if it satisfies at least one literal in each of F's clauses. An assignment satisfies a positive (resp. negative) literal if the corresponding variable is assigned the value $true$ (resp. $false$).

A SAT solver is a tool which solves a given SAT instance. The state of the art technique used in modern SAT solvers is *conflict driven clause learning (CDCL)* [4], which is an extension of the DPLL algorithm [5]. One of the key procedures of CDCL (and DPLL) is *unit propagation (UP)*. Unit propagation calculates variable assignments which are directly implied by a given partial assignment.

As input, UP takes a partial variable assignment β and a formula F_{cnf}. The UP algorithm then iterates over all unit clauses (clauses with exactly one literal) in F_{cnf} and checks for each unit clause whether the corresponding literal is part of any other clause in F_{cnf}. If a clause contains the corresponding literal, that clause is satisfied and does not have to be considered anymore. It can be removed from the clause set. If the corresponding literal is negated part of any clause, the negated literal will never evaluate to true and thus can be removed from the clause. Removing literals from a clause can produce new unit clauses which will subsequently be processed by UP as well. The algorithm terminates when there are no unit clauses left or a conflict is detected.

2.0.2 *Product Configuration*

A *configuration task* T is defined as the triplet (V, D, C) where $V = \{v_1, v_2, ..., v_n\}$ represents a finite set of finite domain variables and $D = \{D_1, D_2, ..., D_n\}$ represents a set of values corresponding to the variables (i.e., $dom(v_1) = D_1$). Furthermore, $C = P_{KB} \cup C_R$ represents constraints, where P_{KB} represents the product knowledge and C_R represents a set of user requirements (both represented as CNF formulas). A *configuration* is a selection of values for all variables in the product model. Given a set of variables $V = \{v_1, ..., v_n\}$ and their corresponding domains $D = \{D_1, D_2, ..., D_n\}$, a configuration is an instantiation

$I = \{v_1 = \{i_{11}, ...\}, v_2 = \{i_{21}, ...\}, ..., v_n = \{i_{n1}, ...\}\}\}$ where i_{kj} is an element of $dom(v_k) \in D$. A configuration is called partial, if it does not contain a selection for all variables, otherwise it is called complete. A configuration is called valid when it is complete and consistent, as that the assignments do not contradict the product model.

A product configurator assists the user during the selection of domain values in order to configure a product which fits the user's requirements best. A single selection of a domain value is called a *user decision* [11]. A product configurator takes these user decisions as an input and checks whether they contradict any constraints in the product model. The set of user decisions can be given to the configurator either all at once or one decision at a time, the later is called *interactive configuration*. In an interactive configuration process, the user selects each domain one by one and receives instant feedback about the validity of the configuration after each selection. In addition to checking the validity of the user's selections, an interactive product configurator can mark domain values during the configuration process that would lead to a contradiction and thus to an invalid configuration. This allows the user to avoid contradictions during the configuration process completely and, thus, can decrease the time needed to finish the configuration process. The domains that do only contain values that do not lead to a contradiction are called the set of *valid domains*.

Calculating valid domains using satisfiability as the consistency criteria is an NP-hard problem [8]. This means that after each configuration step an NP-hard problem would have to be solved, in order to update the set of valid domains which would be displayed to the user. But in order to keep response times low, a product configurator can choose to use a less general consistency criteria in order to check, whether a domain value is valid. Using a less general consistency means that the set of valid domains has to be approximated.

Underestimating the set of valid domains means that domain values could be displayed to the user as invalid, even though they would not lead to a contradiction. Underestimating the set of valid domains also means that a user is not able to use the full potential of the configurability of the product and is restricted rather by technical limitations of the product configurator. A configurator which does not underestimate the set of valid domains is called *complete*. Overestimating the set of valid domains means that domain values which would lead to a contradiction could be displayed to the user as valid nonetheless. By selecting such a domain value, a contradiction would arise even though the user would not expect this. A configurator which does not overestimate the set of valid domains is called *backtrack-free* [10] as it guarantees a backtrack-free configuration process.

In this paper we will present an algorithm that overestimates valid domains, therefore the configurator using our method is complete but not backtrack-free.

2.1 Unit-provable Invalid Domains

Overestimating the set of valid domains can also be seen as underestimating the set of *invalid domains*. An invalid domain only contains values which are guaranteed to lead to a contradiction when selected by the user. A possible way to calculate the underestimated set of invalid domains is to only consider invalid domain values which are implied through unit propagation by all previously made user decisions. We call such domain values *unit-provable invalid domain values*. A domain that only contains unit-provable invalid domain values is called a *unit-provable invalid domain*.

Definition 1 (unit-provable invalid domain). *Given the configuration task $T = (V, D, C)$ and a variable $v \in V$, the unit-provable invalid domain $dom^u_{inv}(v)$ for variable v is defined as $dom^u_{inv}(v) = \{d \mid \neg d \in \Gamma^{up}(V, D, C \cup \{v = d\}), \, d \in dom_{unsel}(v)\}, \, v \in V$. Γ^{up} is a configurator which returns a configuration that only contains the user selected domain values and through unit propagation additionally implied domain values. $dom_{unsel}(v)$ refers to the domain values $d \in dom(v)$ for variable v which are currently not selected by the user. Consequently, a unit-provable invalid domain only contains values that the user could have selected in the next step.*

As a side note: Another way of overestimating the set of valid domains is to calculate the set of *1-provable invalid domains*. *1-provability* was developed by Pipatsrisawat and Darwiche [14] and means that it can be shown that a literal l is implied by a CNF formula F_{cnf} by only using unit propagation. In order to show that l is implied by F_{cnf}, $\neg l$ is added as a unit clause to F_{cnf} and then unit propagation is performed. Iff unit propagation results in a conflict then l is implied by F_{cnf}. Consequently, a 1-provable invalid domain only contains values that 1-provably lead to a contradiction. 1-provability is a more general consistency criteria for calculating valid domains than unit-provability and thus the set of 1-provable invalid domains is a superset of unit-provable invalid domains.

In a configurator it is desirable to handle a certain type of conflicts differently when calculating invalid domains. When a user selects a value from a domain with an at-most-one constraint then all the other values become invalid. However, in this case, we make an exception and do not mark the other values invalid. The reason is that the user selecting another value usually intends to change his selection and not attempting to select both values at once. Therefore we can automatically resolve these kinds of conflicts by removing the originally selected value first and then adding the newly selected value.

Next we provide a definition for a unit-provable invalid domain, which considers the previously mentioned type of conflict. We call such a domain an *ar-unit-provable invalid domain*. The prefix *ar* is added in order to indicate that conflicts arising from **a**lternative **r**elations are considered.

Definition 2 (ar-unit-provable invalid domain). *Given a configuration task $T = (V, D, C)$ and a variable $v \in V$, an ar-unit-provable invalid domain $dom^{ar}_{inv}(v)$ for variable v is, except in one case, identical to a unit-provable invalid domain:*

$$dom^{ar}_{inv}(v) = dom^u_{inv}(v) \tag{1}$$

The only case in which a ar-unit-provable invalid domain differs from the unit-provable invalid domain is when the corresponding variable v (1) is a max-one variable (there is an at-most-one constraint bound to its domain) and (2) a value is already assigned to v by the user. In this case, $dom^{ar}_{inv}(v)$ only contains invalid domain values that stay unit-provable invalid even if v would not be a max-one variable but could be assigned multiple values.

The ar-unit-provable invalid domain of a max-one variable v thus only contains invalid values that would stay invalid even if the user would manually deselect the already assigned value to v before making his new decision. For any other variable, there is no value that the user could deselect before making a new decision in order to avoid an obvious but undesired conflict. Thus all other ar-unit-provable invalid domains are identical to unit-provable invalid domains.

Example 1 (ar-unit-provable invalid domains). *Consider a product model consisting of the variables $V = \{v_1, v_2\}$ with domains*

$dom(v_1) = \{d_1, d_2, d_3\}$ and $dom(v_2) = \{d_4, d_5, d_6\}$, where v_1 is a max-one variable and a single constraint: d_4 excludes d_2. The user assigns d_1 to v_1 and d_4 to v_2. The formula F_{cnf} represents the product model. Performing unit propagation on F_{cnf} with $\beta = \{d_1, d_4\}$ as the initial assignment leads to the implied negative literals $\{\neg d_2, \neg d_3\}$. Those negative literals are then extracted from the solution and since they correspond to a domain value in the product model, they form the unit-provable invalid domain $dom^u_{inv}(v_1) = \{d_2, d_3\}$ for v_1. The unit-provable invalid domain for v_2 is empty. But since v_2 is not a max-one variable, its ar-unit-provable invalid domain is empty as well. v_1, on the other hand, is a max-one variable and was assigned a value by the user. Thus, its ar-unit-provable domain differs from its unit-provable invalid domain. In order to determine the ar-unit-provable invalid domain for v_1, the domain value assigned to variable v_1 is removed from the configuration and unit propagation is performed again with the initial assignment $\beta = \{d_4\}$. Only d_2 is negatively implied again and thus forms the ar-unit-provable invalid domain $dom^{ar}_{inv}(v_1) = \{d_2\}$ of v_1.

Definition 2 can be generalized to *ar-invalid domains* and also to *ar-valid domain* by using satisfiability as the consistency criteria instead of only considering domain values that are implied by the currently selected values and unit propagation.

3 Related work

In this Section, we will look at related work for calculating the set of (in)valid domains. Methods for calculating the set of valid domains can be generally grouped into online and offline methods. The goal of offline methods is to solve the hard part (NP-hard) of calculating valid domains before the interactive configuration process is started. This in turn allows to check whether a domain value is valid in polynomial time during the configuration process.

There are three main kinds of offline methods in the literature. The first method is the translation of the configuration problem into a binary decision diagram (BDD) [15, 8]. A binary decision diagram is a graph representation of the configuration problem, which allows to decide whether a partial configuration can be extended to a valid configuration in linear time in relation to the amount of graph nodes.

The second and third approach rely on the representation of the configuration problem as a constraint satisfaction problem (CSP). Checking whether a partial configuration can be extended to a valid configuration corresponds to checking whether the sub-CSP induced by the corresponding partial variable assignment is satisfiable. As checking satisfiability is NP-complete, researchers have developed several preprocessing methods for CSPs [7, 6]. The most relevant type of preprocessing methods are decomposition methods. Decomposition methods determine sub problems of the original problem to which all solutions are computed and the gained insights are used to transform the original problem into an intermediate representation [18]. This approach can guarantee that a solution to a CSP can be found in polynomial time [7], as long as the problem size does not grow superpolynomially during the preprocessing.

Adaptive consistency is a decomposition method for preprocessing CSPs. By ensuring adaptive consistency, a solution can be found in polynomial time by guaranteeing a backtrack-free solution calculation when given a fixed variable assignment order. Adaptive consistency is ensured by adding additional constraints to the CSP and finding a solution to it in polynomial time is, however, only possible if adaptive consistency does not increase the size of the CSP superpolynomially [7].

Beck et al. [3] describe an algorithm that ensures adaptive consistency but without possibly adding a super polynomial amount of new constraints to the CSP. They do this by removing domain values from domains instead of adding additional constraints. But this approach leads to solution loss. This means (in terms of product configuration) that a customer cannot choose a certain configuration even though it would have been a valid one. Beck et al. [3] argue that loosing valid configurations can be acceptable in some cases and thus there is a trade-off between space complexity and solution loss.

```
   Input: configuration task $T = (V, D, C)$
1  Initialize valid domains $dom_{val}(v) = \emptyset$ for all $v \in V$
2  foreach unselected domain value d, $d \in dom(v)$ do
3      if $SAT(T$ extended with $d)$ then
4          Add $d$ to $dom_{val}(v)$
5      end
6  end
7  return $dom_{val}(V)$
```

Algorithm 1: Naive approach for calculating valid domains

Online methods do not do any preprocessing but solve an NP-hard problem after each configuration step in order to guarantee a backtrack-free configuration process. Algorithm 1 shows the pseudo code for calculating the set of valid domains using satisfiability as the consistency criteria. To achieve an approximation we can just use unit propagation instead of SAT solving (at line 3 of Algorithm 1), which is the basic idea behind the algorithm used in the commercial product configurator Merlin [2] and also our new algorithm.

4 Approximating Invalid Domains

Our algorithm calculates the ar-unit-provable invalid domains. In the following, we assume that the product model is a Boolean formula F_{cnf} and the variable assignment which corresponds to the user selected domain values is denoted with β. We call the literals in β the *root literals*.

Given a domain value d which is implied by β when β is used as the initial assignment for F_{cnf}, there may exist several subsets $\{S_1, S_2, ..., S_n\}, S_i \subseteq \beta$ which would also imply d if S_i would be used as the initial assignment instead of β. If S_i is minimal, then we call S_i an *implication possibility* of d. For each S_i there is a minimal subset $C \subseteq F_{cnf}$ of clauses such that C implies d through unit propagation when using S_i as the initial assignment. We define two types of equality between two implications possibilities S_1, S_2: **Type-1:** Two implication possibilities S_1, S_2 are equal, iff their corresponding minimal clause sets C_1, C_2 are equal (denoted by $S_1 \overset{C}{=} S_2$) and **Type-2:** Two implication possibilities S_1, S_2 are equal, iff the literals in both sets are equal (denoted by $S_1 = S_2$). It is easy to see that $S_1 \overset{C}{=} S_2$ implies $S_1 = S_2$, but not the other way around.

Example 2 (Implication possibility). *Given $F_{cnf} = (\neg d_1 \vee d_2) \wedge (\neg d_2 \vee d_4) \wedge (\neg d_1 \vee \neg d_3 \vee d_4)$ an initial assignment $\beta = \{d_1, d_2, d_3\}$, and a literal $d_4 \notin \beta$. Then, three implication possibilities for d_4 exist: $S = \{S_1 = \{d_1\}, S_2 = \{d_2\}, S_3 = \{d_1, d_3\}\}$.*

Any literal which is not in the intersection of all implication possibilities of l can be removed from β and l will remain implied. Contrary, removing any literal from the intersection results in l being no longer implied. We call the intersection of all implication possibilities of l the *required root literals* of l, denoted by $req(l)$.

```
Input: configuration task $T = (V, D, C)$, required root
       literals $req(D)$
   $F$ = product model formula
   $\beta$ = initial assignment corresponding to the user decisions
1  $\forall v \in V$ calculate unit-provable invalid domains $dom^{u}_{inv}(v)$
     using $F$ and $\beta$
2  $\forall v \in V$ Initialize ar-unit-provable invalid domains
     $dom^{ar}_{inv}(v) = dom^{u}_{inv}(v)$
3  foreach user selected domain value $d$ that is assigned to a
     max-one variable $v$ do
4      foreach $\hat{d} \in dom^{u}_{inv}(v)$ do
5          if $d \in req(\hat{d})$ then
6              Remove $\hat{d}$ from $dom^{ar}_{inv}(v)$
7          end
8      end
9  end
10 return $dom^{ar}_{inv}(V)$
```
Algorithm 2: Calculating ar-unit-provable invalid domains using required root literals

Now, with the definition of required root literals at hand, we can connect ar-unit-provable invalid domain values to required root literals.

Theorem 1. *Given a Boolean formula F_{cnf}, an initial assignment β, a variable $v \in V$ with domain $dom(v) = \{d_1, ..., d_n\}$ and $d_k \in \beta$ with an arbitrary but fix $k \in [1, n]$ which is assigned to v. If v is a multivalue variable (variable which can be assigned multiple values) or v was not assigned by the user, then $dom^{ar}_{inv}(v)$ and $dom^{u}_{inv}(v)$ are identical. On the other hand, if v is a max-one variable and was assigned by the user then a domain value $d \in dom^{u}_{inv}(v)$ is ar-unit-provable invalid if and only if $d_k \notin req(d)$.*

Proof. Assume that v is a max-one variable and was assigned the value l by the user. A domain value $d \in dom^{u}_{inv}(v)$ is also ar-unit-provable invalid iff it stays negatively implied even if the user would deselect l. Deselecting l means that l is removed from β. Deciding whether d is ar-unit-provable invalid thus means that it has to be checked whether $\neg d$ stays implied if l is removed from β. A required root literal of $\neg d$ is a literal that has to be part of β in order for $\neg d$ to be implied. So if l is a required root literal of $\neg d$, and the user would deselect l, $\neg d$ would not stay implied and thus d would not be ar-unit-provable invalid. If l is not a required root literal of $\neg d$, and the user would deselect l, $\neg d$ would stay implied and thus d would be ar-unit-provable invalid.

If v was not assigned by the user or is not a max-one variable, then $dom^{ar}_{inv}(v)$ and $dom^{u}_{inv}(v)$ are identical by definition 2. □

Algorithm 2 makes use of Theorem 1 for calculating the set of ar-unit-provable invalid domains. First, the unit-provable invalid domains are calculated (line 1). In line 2, the ar-unit-provable invalid domains are initialized with the unit-provable invalid domains. The foreach loop in line 3 iterates over all user assigned max-one variables to check whether the previously calculated unit-provable invalid domain values of those variables are also ar-unit-provable invalid. The foreach loop in line 4 iterates over each unit-provable invalid domain value of v in order to check whether each individual value is ar-unit-provable invalid. In line 5, it is checked whether d is a required root literal of $\hat{d}$. If this is the case, $\hat{d}$ is not ar-unit-provable invalid and removed from $dom^{ar}_{inv}(v)$.

4.1 Deciding whether a Literal is a Required Root Literal

The general approach is to create an implication graph during unit propagation which allows us to retrace by which literals each literal was implied. This consequently also allows us to calculate which literals of the initial assignment lead to the implication of certain other literals. Note that calculating the complete set of required root literals for each literal is not necessary in order to correctly execute Algorithm 2. Checking whether d is part of the required root literals of $\hat{d}$ can be performed by using the implication graph to enumerate all type-1 distinct implication possibilities of $\hat{d}$ until one is found that does not contain d. If such an implication possibility was found, it has been proven that d is not a required root literal of $\hat{d}$. The reason is that d has to be contained in all implication possibilities $S_1, ..., S_n$ in order to be in the intersection $S_1 \cap ... \cap S_n$.

An implication graph is a directed acyclic graph and represents which literals implied which other literals during unit propagation [1]. In order to build the implication graph during unit propagation, each time a clause becomes unit, a pointer to that clause is stored and associated with the implied literal. Root literals are initially not associated with any clause. When unit propagation is finished, the saved clauses and associated literals can be interpreted as the implication graph. A node corresponds to a single implied/root literal. An edge corresponds to the implication of a literal. More precise: An edge from literal k to literal l exists, if there exists a clause c with $k \in c$ and c became unit so that l was implied. The edge is labeled with c accordingly. The resulting implication graph is, however, dependent on the order in which the literals were propagated and also does not contain all the possible ways a certain literal can be implied. What we need is a *full implication graph*[1]. Van Gelder [17] also developed a method to capture all possible implications of any literal l but Abrame et al. were the first to come up with the notion of a full implication graph.

Definition 3 (Full implication graph). *Adapted from [1]. Let F_{cnf} be a CNF formula defined on a set of Boolean variables X and let I be a partial assignment. A full implication graph is a directed graph $G = (V_v, V_{uc}, E)$ where V_v is the set of nodes which represent the assigned variables, V_{uc} is the set of nodes which represent the unit clauses and E is the set of arrows which link the unit clauses (the predecessors) to their propagated variables and also link the assigned variables to the clauses they reduce (the successors).*

The full implication graph is built in the same way during unit propagation as the conventional implication graph with the difference that satisfied clauses must not be ignored.

4.1.1 Searching in the Full Implication Graph

We developed an algorithm that considers each implication possibility one by one by performing a depth first search on a full implication graph. However, instead of calculating each implication possibility individually, our algorithm makes use of theorem 2 to directly decide for each traversed literal, whether l is a required root literal of that literal.

For a literal x let C be the clause which became unit during unit propagation and implied x. By *predecessor clause* we mean the set of literals $\overline{C \setminus \{x\}}$. Example: for $(\neg a \vee \neg b \vee x)$ and x the predecessor clause would be $\{a, b\}$.

Theorem 2. *Let $C = \{C_1, ..., C_n\}$ be a set predecessor clauses of an arbitrary literal x. An arbitrary root literal l is not a required root* *literal of a predecessor clause $C_i = \{v_{i1}, ...\}$, iff no literal $v_{ij} \in C_i$ has l as a required root literal. If at least one literal $v_{ij} \in C_i$ exists, that has l as a required root literal, then l is a required root literal of C_i as well. l is not a required root literal of x, iff at least one predecessor clause C_i exists that does not have l as a required root literal. On the other hand, l is a required root literal of x, iff all predecessor clauses of x have l as a required root literal. In the case that C is empty (i.e., x has no predecessor clauses and thus is a root literal), l is not a required root literal of x iff $l \neq x$. l is a required root literal of x, iff $l = x$.*

Proof. Follows from the definition of required root literals. □

Theorem 2 is recursively applied to each traversed literal during the depth first search in the full implication graph in order to eventually decide whether l is a required root literal of k.

The algorithm (see Algorithm 3 for a pseudo-code) iterates over all predecessor clauses of k. First, it is assumed, that l is not a required root literal of the currently processed predecessor clause C. Then we iterate over each literal in C in order to falsify this assumption. This is done by recursively calling the algorithm in order to evaluate whether l is a required root literal of the currently processed literal. If this is the case, the assumption was falsified and it was proven that l is also a required root literal of C. Consequently, no more literals in C have to be evaluated and the next predecessor clause can be processed. If the assumption could not be falsified, then it was proven that at least one predecessor clause of k exists where all literals do not have l as a required root literal and thus k does not have l as a required root literal. Consequently, the algorithm returns $false$. If all predecessor clauses have l as a required root literal, the algorithm cannot return early and has to iterate over all predecessor clauses. In this case, l is a required root literal of k and $true$ is returned. The termination condition of our recursive algorithm is either that k is a root literal and we return (according to Theorem 2) $false$ if k is not equal to l. If, on the other hand, k equals l, the algorithm cannot return $true$ immediately because k could have predecessor clauses since root literals can imply other root literals.

Contrary to a conventional implication graph, a full implication graph can contain cycles [1]. A cycle in the full implication graph would have the effect that our above described algorithm traverses the same set of clauses and literals indefinitely. In order to guarantee, that our algorithm terminates, cycles have to be detected and avoided. That is done by simply maintaining the set of visited literals. In the following, we prove the the correctness of our algorithm.

Theorem 3. *Algorithm 3 can correctly decide if l is a required root literal of x, even if predecessor clauses that contain cycles are skipped during the search.*

Proof. Let x be the literal that was assigned to parameter k in the initial call to the algorithm. Each implication possibility of x is associated with a distinct set of clauses. This set of clauses is sufficient in combination with the corresponding set of root literals to imply x through unit propagation. Since this set of clauses is minimal, each literal is implied only once during unit propagation. Consequently, the resulting full implication graph is acyclic. A correct solution can be found for x when searching in an acyclic full implication graph since the clause set that belongs to an implication possibility is a subset of the original clause set. The resulting full implication graph of that subset is also a sub graph of the original full implication graph. Further, each full implication graph induced by each individual implication possibility is contained in the original full implication graph

```
1  isReqRootLit ( literal $k$, root literal to check $l$, visited
      literals $Visited$, all root literals $\beta$, predecessor clauses for
      each literal $Predecessors$)
2  if $k \in \beta$ then
3      if $k \neq l$ then
4          return $false$
5      end
6  end
7  $Visited = Visited \cup k$
8  foreach $C \in Predecessors(k)$ do
9      if $C \cap Visited \neq \emptyset$ then
10         continue
11     end
12     $isNotRequired = true$
13     foreach $j \in C$ do
14         $irl$ = isReqRootLit ($j$, $l$, $Visited$, $\beta$, $Predecessors$)
            if $irl = uncertain$ then
15             $isNotRequired = false$
16             break
17         end
18     end
19     if $isNotRequired$ then
20         $Visited = Visited \setminus k$
21         return $false$
22     end
23 end
24 $Visited = Visited \setminus k$
25 if $Visited = \emptyset$ then
26     return $true$
27 end
28 return $uncertain$
```

Algorithm 3: is l a required root literal of k? Cycles are handled

as well. Thus, the algorithm can find each implication possibility in the original full implication graph by traversing the original full implication graph in the correct order. The algorithm is thus also able to traverse all induced full implication graphs with a single continuous acyclic search order. With a known acyclic continuous order, the algorithm would be able to reconstruct all implication possibilities of x, which in turn allows the algorithm to prove that l is not a required root literal. Thus, skipping cycles during the search has no effect on whether the algorithm is able to find a correct result. □

This proof requires that Algorithm 3 traverses the implication graph in a certain order. But since it is not possible to know this order beforehand, each possible order has to be tried. The algorithm chooses the next clause or literal to process by whichever clause or literal was not tried yet. This in turn can have the effect that the algorithm selects a wrong search order and thus reaches a dead end where only cycles are detected or a different order has to be tried in order to find additional implication possibilities. In such a case, the algorithm has to backtrack. For this reason, literal k is removed from $Visited$ in line 20 and 24. This allows the algorithm to process the same literal again at a later point but in the context of a different order. However, the downside of this approach is, that it requires an exponential amount of operations in the worst case.

If all different orders have been tried without being able to prove

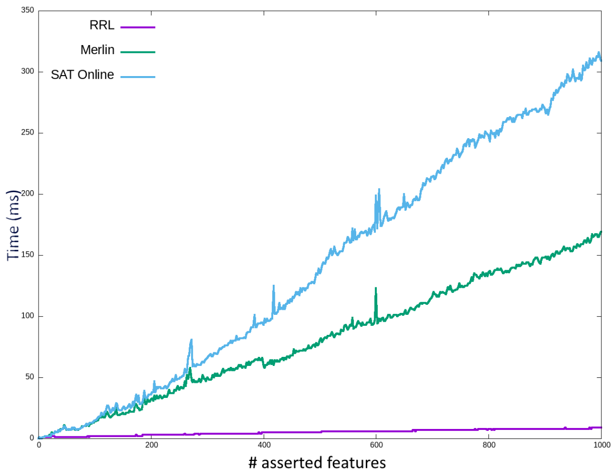

Figure 1: Runtime evaluation results.

that l is not a required root literal of x, the algorithm returns $true$. In line 25, it is checked whether $Visited$ is empty. If this is the case, the currently processed literal is x. Since the check is performed after each predecessor clause was processed, it can be concluded that all different orders have been tried and we return $true$.

On a last note regarding cycles: The algorithm is also able to correctly determine that l is a required root literal of k, if no cycles were encountered at all while processing all predecessor clause of k and the corresponding literals. In such a case, the algorithm would not have to return $uncertain$ but could return $true$ instead. But since the return value $true$ would be handled the same way as $uncertain$ is, we do not differentiate between those two return values and stick to $uncertain$.

4.1.2 Optimizing the Search

So far, we have discussed how the algorithm works. But due to cycles in the full implication graph, the algorithm is required to perform en exponential amount of operations in the worst case. In this section, we will present two optimizations that minimize the amount of operations that the algorithm has to perform.

Optimization 1: It is sufficient to find a single predecessor clause of literal k where each literal does not have l as a required root literal in order to prove that k does not have l as a required root literal as well. Since evaluating k at a later point during the search again would yield the same result, this result can be saved. The next time the algorithm encounters k, the saved result can be used instead of having to search the full implication graph again.

Optimization 2: Algorithm 2 uses Alorithm 3 to decide whether $\hat{d}$ is a required root literal of d. The set of literals that do not have $\hat{d}$ as a required root literal can be passed to Algorithm 2 as parameter. This set can be reused between independent calls to Algorithm 2 as long as the literal assigned to parameter $\hat{d}$ is the same. This is the case in Algorithm 2 for each call to Algorithm 3. This way, each subsequent call to Algorithm 3 terminates faster since the evaluation result of more literals is already known.

5 Experimental Evaluation

In this Section we present the results of our experimental evaluation of Algorithm 2 regarding its runtime and accuracy.

As benchmarks we will use feature models as product models. Mendonca provides a repository of real-world and generated feature models [12]. One kind of generated models – so called *3-CNF Feature Models* (3-CNF FM) tend to induce much harder problems than most of realistic models [13], therefore we will use those for our performance evaluation. To translate the feature models to CNF we used the library provided by Mendonca [12]. We used the 3-CNF FM generator with its default parameters except for the number of features (which we increased to 10,000 instead of 100) and the percentage of features considered for cross tree constraints (increased to 50 instead of 20) in order to generate harder feature models.

For the accuracy evaluation we used five real world feature models [12], namely: SubseaControlSystem, ubuntu1410, Ubuntu12, Self-Generated and Big Data System. These feature models are significantly smaller than the generated models, containing only a few hundred features.

For the performance evaluation experiments we will use the following algorithms. Each implemented by us in Java.

- **Naive algorithm**: The naive approach for calculating the ar-unit-provable invalid domains – running unit propagation for each candidate separately.
- **Merlin algorithm**: An algorithm used by the commercial product configurator Merlin. It is an optimized variant of the naive approach that reduces redundant unit-propagations.
- **RRL algorithm**: The algorithm introduced in this paper – Algorithm 2.

For our accuracy experiments, we will use the RRL algorithm to calculate the ar-unit-provable invalid domain values, the naive approach for calculating the ar-1-provable invalid domain values and the adapted SAT Online algorithm (Algorithm 1) for calculating ar-invalid domain values.

All experiments were performed on a Computer with Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GHz, 8GB RAM, NixOS 20.03 operating system and Java 11 (OpenJDK).

Figure 1 shows the results of the performance measurement. The naive algorithm is clearly the slowest one and needs 135ms for 500 asserted features and 309ms for 1,000 asserted features. The second slowest algorithm is Merlin's algorithm which needs 81ms at the 500 mark and 169 at the 1,000 mark. The RRL algorithm needs 5ms at the 500 mark and 9ms at the 1,000 mark. This corresponds to a speedup at the 500 and 1,000 mark of roughly 18 compared to the Merlin

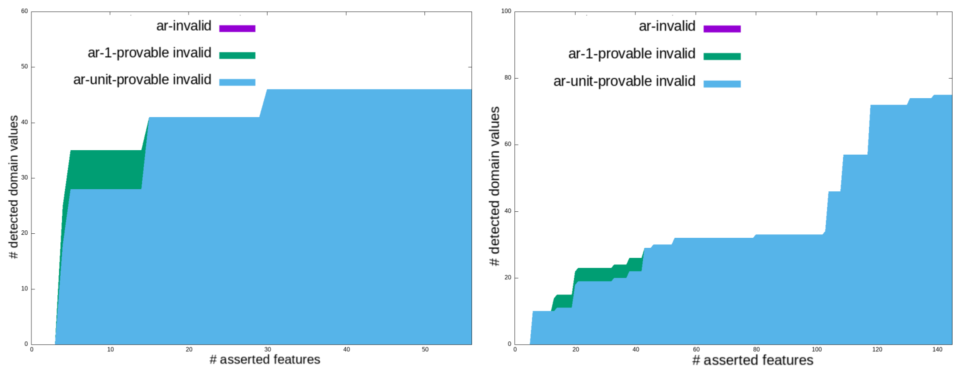

Figure 2: Size measurements for feature models *SubseaControlSystem* (left) and ubuntu410 (right).

algorithm and a speedup of roughly 34 compared to the naive algorithm.

Our algorithm has in the worst case an exponential time complexity, But it still outperforms the naive and Merlin's algorithm even though they have a linear worst case time complexity. We argue, that the reason for that is, that most traversed literals evaluate to $false$. This allows the algorithm, due to our optimization to reuse evaluation results instead of having to search the implication graph again at a later point. Another probable reason is that the implication graph most likely never has to be traversed completely in order to decide whether a literal is a required root literal.

Figure 2 shows the measured sizes of the set of ar-unit-provable invalid domain values, ar-1-provable invalid domain values and ar-invalid domain values for the *SubseaControlSystem* and *ubuntu1410* feature models. For the other three feature models the set of ar-unit-provable invalid domain values is identical to both the set of ar-1-provable invalid domain values and ar-invalid domain values.

For the *SubseaControlSystem* the set of ar-invalid domain values is identical to the set of ar-1-provable values. The set of ar-1-provable invalid domain values differs from the set of unit-provable invalid domain values in 10 of the 60 test cases, the maximum difference in size is 7. In case of *ubuntu1410_* the set of ar-1-provable invalid domain values differs from the set of unit-provable invalid domain values in 29 of 150 test cases, the maximum difference in size is 4.

The results indicate that the ar-unit-provable invalid domain values are most of the time identical to the ar-invalid domain values. Only in two feature models additional ar-1-provable invalid domains were detected. We do believe that our results are representative for small feature models with few constraints. But due to the available tools and our limited time frame we were not able to measure big feature models with many constraints.

6 Conclusion

The challenge of calculating valid domains is that this problem is NP-hard but we need to calculate them for hundreds of variables within milliseconds. For this reason, we developed an algorithm that approximates the valid domains. A further challenge is to consider the automatic resolution of conflicts that arise from selecting a second value within an alternative relation as domain values could become valid that would be invalid otherwise.

We developed our algorithm based on the idea of required root literals. Our algorithm has to perform unit propagation only once and then relies on the full implication graph for further calculations. Even though this approach induces an exponential worst case complexity, we could not observe it in our experiments. Instead, we found that the calculation time of our algorithm appears to grow linearly with the the amount of asserted features. In future work, the average case complexity of our algorithm should be determined and it should be analyzed, under which circumstances the algorithm might require an exponential computation time.

We additionally measured the accuracy of the approximation achieved by our method. We found that for small feature models with few constraints this approximation is very accurate. These results are, however, not transferable to large feature models with many constraints. Additional measurements have to be performed in future work in order to get a better understanding of the accuracy.

REFERENCES

[1] Andre Abrame and Djamal Habet, 'Handling all unit propagation reasons in branch and bound max-sat solvers', *Proceedings of the 7th Annual Symposium on Combinatorial Search, SoCS 2014*, 2–9, (2014).

[2] CAS Software AG. Product Configurator Merlin. `https://www.cas-merlin.de`, May 2021.

[3] J Christopher Beck, Tom Carchrae, Eugene C. Freuder, and Georg Ringwelski, 'A SPACE-EFFICIENT BACKTRACK-FREE REPRESENTATION FOR CONSTRAINT SATISFACTION PROBLEMS', *International Journal on Artificial Intelligence Tools*, **17**, 703–730, (2008).

[4] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability*, 2009.

[5] Martin Davis, George Logemann, and Donald Loveland, 'A Machine Program for Theorem-Proving', *Communications ACM*, **5**, 394–397, (1962).

[6] Eugene C. Freuder, Tom Carchrae, and J. Christopher Beck, 'Satisfaction guaranteed?', *Anaesthesia*, **67**, 924–925, (2012).

[7] Georg Gottlob, Nicola Leone, and Francesco Scarcello, 'Comparison of structural CSP decomposition methods', *Artificial Intelligence*, **124**, 243–282, (2000).

[8] T Hadzic and H R Andersen, 'An introduction to solving interactive configuration problems', *Technical Report. TR-2004-49.*, (August), (2004).

[9] Mikoláš Janota, 'Do sat solvers make good configurators?', in *SPLC (2)*, pp. 191–195, (2008).

[10] Mikoláš Janota, *SAT solving in interactive configuration*, Ph.D. dissertation, University College Dublin, 2010.

[11] Mikoláš Janota, Goetz Botterweck, Radu Grigore, and Joao Marques-Silva, 'How to Complete an Interactive Configuration Process?', *Theory and Practice of Computer Science*, 528–539, (2010).

[12] Marcilio Mendonca, Moises Branco, and Donald Cowan, 'Splot: software product lines online tools', in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pp. 761–762, (2009).

[13] Marcilio Mendonca, A Wasowski, and Krzysztof Czarnecki, 'SAT-based analysis of feature models is easy', *Proceedings of the 13th International Software Product Line Conference*, 231–240, (2009).

[14] Knot Pipatsrisawat and Adnan Darwiche, 'On the power of clause-

learning SAT solvers as resolution engines', *Artificial Intelligence*, **175**, 512–525, (2011).

[15] Antoine Rauzy, 'A brief introduction to Binary Decision Diagrams', *Journal Europeen des Systemes Automatises*, **30**, 1033–1050, (1996).

[16] Mitchell M. Tseng, Jianxin Jiao, and M. Eugene Merchant, 'Design for mass customization', *CIRP annals,*, **45(1)**, 153–156, (1996).

[17] Allen Van Gelder, 'Generalized conflict-clause strengthening for satisfiability solvers', *Lecture Notes in Computer Science*, **6695 LNCS**, 329–342, (2011).

[18] Rainer Weigel and Boi Faltings, 'Compiling constraint satisfaction problems', *Artificial Intelligence*, **115**(2), 257–287, (1999).