Do SAT Solvers Make Good Configurators?

Mikoláš Janota
Lero, School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland
mikolas.janota@ucd.ie

Abstract

A configuration process *is about finding a* configuration*, a setting, that satisfies the requirements given by the user and constraints imposed by the domain. Feature models are used to record product domains and constraints imposed on individual products. As such constraints are in practice of complex nature, it is desirable to perform the configuration* interactively. *This article shows how to utilize a SAT solver in an interactive configuration process to provide support to the user.*

1. Introduction

Configurable complex systems have gained attention in recent years in such domains as the automotive industry and *feature models* are used to capture the intended product domain [7, 11].

Each feature corresponds to a distinctive aspect of a product and a customer specifies his requirements by defining the desired features. Features that were not specified by the customer, must be *configured* at the producer's side. When the number of features is large and dependencies between them are complex, this represents non-trivial tasks for the humans involved.

This is a clear motivation for *interactive* support during the configuration process — a tool provides hints or performs decisions throughout the whole process, alleviating the burden imposed on the human.

Other researchers targeted this problem before. Batory [1] uses the so-called *logic truth maintenance system*, whose disadvantage is that the approach does not guarantee *backtrack-freenes* — the user may hit a dead end during the configuration process.

Other approaches [10] use Binary Decision Diagrams (BDDs) [3] that guarantee backtrack-freenes but are known to suffer from space explosion as they are explicit representations of all possible configurations.

This article outlines a third path, building mainly on Batory's work, and that is it to apply a *SAT solver*, a tool for deciding the satisfiability of Boolean formulas (see e.g., [8, 5]).

The motivations to use of a SAT solver is two-fold. Firstly, nowadays SAT solvers are extremely efficient despite the NP-completeness of the problem and still improving as illustrated by the yearly *SAT Competition*[1]. Secondly, a SAT solver is a step towards a more general *constraint solver* with the support for other than merely Boolean domains.

To the knowledge of the author guaranteeing backtrack-freeness of and interactive configuration process by using a SAT solver is novel.

2. Background

Feature models [7] play an important role in *Software Product Line Engineering* [4]. In this article we are concerned only with feature models as descriptions of the product domain (also called *problem space*). In particular, we only need to know the semantics of the model that we are dealing with.

Various semantics of feature models exist, depending on the their expressiveness [9, 6]. As we will be dealing with configuring a feature model, it is natural to use so-called *constraint satisfaction problem* (CSP) as the structure capturing the semantics; the following definition introduces the concept.

Definition 2.1 *A* constraint satisfaction problem *is a triple* $\langle X, D, \mathcal{C} \rangle$*, where* $X \equiv x_1, \ldots, x_n$ *are variables,* $D \equiv D_1, \ldots, D_n$ *are their respective domains and* $\mathcal{C} \subseteq D_1 \times \cdots \times D_n$ *is a* constraint.

A valuation *is a function from variables to their domains, a valuation is called* partial *iff not all variables from* X *are assigned a value, it is called* complete *iff all variables from* X *are assigned a value.*

[1] http://www.satcompetition.org/

A solution v *to a constraint satisfaction problem is a complete valuation such that* $\langle v(x_1), \ldots, v(x_n)\rangle \in \mathcal{C}$.

The following toy example illustrates how feature models map to CSP, for more details on this topic see for instance the work of Benavides et al. [2].

Example 2.1 *A software application is developed for two markets: for Ireland and United States. The application deals with measurements and hence supports inches and centimeters. When released to Irish market, centimeters must be used whereas inches must be used for United States. The corresponding CSP has two variables* $Country$ *and* $Units$ *whose domains are* $\{IE, USA\}$ *and* $\{cm, in\}$*, respectively. The constraint* $\mathcal{C}$ *comprises the following tuples (pairs in this case).*

$$\{\langle IE,\ cm\rangle, \langle USA,\ in\rangle\}$$

During the *configuration process* the user makes decisions by assigning values to variables and his goal is to find a solution to some given CSP.

In this article we will consider the scenario when a tool, let us call it a *configurator*, reacts upon every user's decision and provides some form of feedback. More specifically, we will be concerned with configurators that disable such values from variable domains whose selection would prevent finding a solution to the CSP.

In our small example, when the user selects Ireland as the desired country, the inches get disabled and centimeters selected automatically as that is the only option left.

The configurator is called *backtrack-free* iff it enables only the values for which there exists a solution, meaning that the user will never reach a dead end forcing him to retract some previous decisions. A configurator is called *complete* iff it does not disable values for which there exists a solution.

Formally, in each step of the configuration process, there is a user selection s_u — a valuation on variables $X_u \subseteq X$. The configurator is complete and backtrack-free iff for any $x_i \notin X_u$, the configurator disables the value v from the domain D_i if and only if there is no solution to the CSP such that it agrees with s_u on the variables X_u and assigns v to x_i.

2.1. Propositional World

A *propositional constraint satisfaction problem* is such CSP where each of the variable domains is $\{\text{TRUE}, \text{FALSE}\}$. This makes the job of the configurator somewhat simpler. If the configurator infers that a solution exists for only one of the two values, the variable must assume the second value and the user will not be allowed to change it; we will say that such variable is *locked*.

Note that if there is no solution for either of the two values for some variable, the given CSP does not have a solution as all variables must be assigned to by a solution.

Rather unsurprisingly, any propositional constraint satisfaction problem $\langle X, D, C\rangle$ can be captured as a Boolean formula on the variables from X such that the valuations satisfying the formula correspond to the solutions of the problem.

Example 2.2 *Even though Example 2.1 is not given as propositional, it can be easily modeled as such. The following formula illustrates the principle.*

$$(IE \vee USA) \wedge (\neg\, IE \vee USA) \wedge IE \Rightarrow cm \wedge (in \vee cm) \wedge (\neg\, in \vee \neg\, cm) \wedge USA \Rightarrow in$$

As we are interested in the existence of solutions, it means that we are interested in satisfying a Boolean formula. To this end we will utilize a *SAT solver*. A SAT solver accepts as input a formula in so-called *conjunctive normal form* (CNF). A formula is in CNF iff it is a conjunct of disjunctions of variables or negated variables, e.g., $(\neg a \vee b) \wedge (\neg b \vee c)$. Each of the conjuncts, e.g., $\neg a \vee b$, is called a clause and the dijuncts are called literals (a single literal is also considered as a clause). An important property of CNF is that any Boolean function can be captured in a CNF. Note that formula in Example 2.2 is in CNF after rewriting the implications as disjunctions.

A SAT solver determines whether the given formula is satisfiable or not. If it is satisfiable, it produces a variable valuation such that the formula evaluates to TRUE. If such valuation does not exist (the formula is unsatisfiable), it produces a proof of the unsatisfiability.

3. A SAT solver in Configuration Process

Our goal here will be to device an algorithm TEST-VARS that is called at the beginning of the configuration process and then after each user's decision.

We will rely on the following armory. For any variable we can call the procedures LOCK and UNLOCK to disable and enable, respectively, the user to set the variable's value. Further, calls to SET($variable$, $value$) sets a value for a variable; RESET($variable$) brings the variable to an unassigned state.

The SAT solver is represented by the function SAT(ψ) that returns either **null** iff ψ is unsatisfiable or it responds with a set of literals (negated or unnegated variables) that correspond to a satisfying valuation of ψ. For example, it may respond $\{a, \neg b\}$ to the query $a \vee b$.

The state of the configuration process and the constraint is captured in the formula ϕ. So for instance, if the CSP under concern is represented by the formula ψ and the user

```
TEST-VARS()
1  foreach x that was not assigned to by the user
2      do CanBeTrue ← TEST-SAT(φ, x)
3         CanBeFalse ← TEST-SAT(φ, ¬x)
4         if ¬(CanBeTrue ∧ CanBeFalse)
            then error "Unsatisfiable constraint!"
5         if ¬CanBeTrue then SET(x, FALSE)
6         if ¬CanBeFalse then SET(x, TRUE)
7         if CanBeTrue ∧ CanBeFalse
8           then RESET(x)
9                UNLOCK(x)
10          else LOCK(x)
```

Figure 1. Basic version

sets the variable v_1 to TRUE and the variable v_2 to FALSE, the formula ϕ will be equal to $\psi \wedge v_1 \wedge \neg v_2$.

Figure 1 presents the skeleton of the algorithm. For each variable it computes whether there are satisfying valuations having the variable set to TRUE and FALSE, respectively. The four possible combinations are investigated: If neither of them exists, then ϕ itself is unsatisfiable (line 4). If exactly one exists, then the variable is enforced to have the value for which there *is* a satisfying valuation (lines 5, 6) and the variable is locked (line 10). If both exist, the algorithm makes sure the variable is in the default state, i.e., unassigned and unlocked (lines 8, 9).

This algorithm calls the solver twice on each variable, so it is warranted to investigate if it can be made more efficient.

First let us look at the situation when the call to SAT returns a satisfying valuation. If a variable x_i in such valuation was assigned a certain value, TRUE let's say, then $\phi \wedge x_i$ is satisfiable (since $\phi \wedge l \wedge x_i$ is satisfiable). Therefore, there is no need to call the solver on $\phi \wedge x_i$.

So the first improvement to the algorithm is to store the values of which we already know that appear in satisfying valuations encountered so far.

Can, on the other hand, the negative response of the solver be useful? Yes it can! Consider the situation when the constraint contains the formula $x_1 \Rightarrow x_2$. If the solver knows that x_2 cannot be TRUE, it can quickly deduce that x_1 cannot be TRUE either[2]. This motivates the second improvement: storing the values that were disabled and conjoing them to the overall formula in subsequent queries.

Figure 2 presents the procedure TEST-SAT. Literal l in inserted into the set $KnownValues$ if $\phi \wedge l$ if satisfiable. Analogously, literal l is added to the formula $DisabledValues$ if $\phi \wedge l$ is unsatisfiable.

[2] The solver detects that from the clause $\neg x_1 \vee x_2$ the literal $\neg x_1$ must be TRUE to satisfy the clause. This technique is known under the name *Unit Propagation* and is an essential part of state-of-the-art solvers.

```
TEST-SAT(φ: Formula, l: Literal) : Boolean
  if l ∈ KnownValues then return TRUE
  if l ∈ DisabledValues then return FALSE
  L ← SAT(φ ∧ l ∧ ⋀_{k∈DisabledValues} ¬k)
  if L ≠ null
    then KnownValues ← KnownValues ∪ L
    else DisabledValues ← DisabledValues ∪ {l}
  return L ≠ null
```

Figure 2. Improved version of TEST-SAT

When and how do we initialize the two sets? The safest (and coarsest) approach is to empty both sets whenever ϕ changes, i.e., at he beginning of each execution of TEST-VARS. However, if we know that ϕ has been strengthened, which happens for instance when a user sets a value of an unassigned variable, we can keep $DisabledValues$ and empty $KnownValues$. Analogously, if ϕ is weakened, e.g., when a user's decision is retracted, we can keep $KnownValues$ and empty $DisabledValues$. For further discussion on this topic see Section 6. To conclude, the following two procedures show how TEST-VARS is invoked.

```
ASSERT-DECISION(l : Literal)
  decisions ← decisions ∪ {l}
  φ ← ⋀_{l∈decisions} l
  KnownValues ← ∅
  TEST-VARS()
```

```
RETRACT-DECISION(l : Literal)
  decisions ← decisions ∖ {l}
  φ ← ⋀_{l∈decisions} l
  DisabledValues ← ∅
  TEST-VARS()
```

3.1. Producing Explanations

For each locked variable a configurator should explain to the user *why* it was locked. The explanation should contain the user's decisions that led to the lock, but even better, the relevant parts of the constraint.

In our case, when the constraint is in a CNF, obvious candidates for parts of the constraint are the individual clauses (disjunctions of literals).

Recall that a value is locked if the solver returns unsatisfiability for the other value. For example, if a variable was locked in the TRUE value, then it must have been shown that there is no solution with the variable having the value

FALSE. The *proof* of the unsatisfiability is exactly the explanation we are looking for.

Obtaining a proof from a SAT solver is rather straightforward, see for example [13]. The proof is a subset of the input clauses whose conjunct is unsatisfiable.

However, a proof obtained from the solver might not be minimal in the sense that removing certain clauses or user-decisions will still yield an unsatisfiable formula. Naturally, for the user-friendliness sake, it is desirable for the explanation to be small.

To this end a fast technique with good results was proposed by Zhang and Malik [12], which calls the solver again on the proof that it has just produced. As the proof is an unsatisfiable formula, the solver will find it unsatisfiable and produce a new proof (possibly smaller than the original one). This process is iterated until the proof remains unreduced by the solver.

4. Implementation

The author implemented the ideas presented in Section 3 and the implementation can be found at our research group's website [3].

Figure 3 offers a screenshot of the application. A subtle difference from the presentation in Section 3. Due to the check-box user interface, that the user can only *select* features — set variables to TRUE— and *deselect* features — set TRUE variables to unassigned. Consequently, user decisions comprise merely non-negated literals.

Apart from automated selections and explanations, the application detects which constraints still remain to be satisfied. In the image we can see that the type of engine and gearshift are yet to be specified. Another functionality is the *configuration completion*, which fills in some values, determined by the SAT solver, that satisfy the unsatisfied constraints.

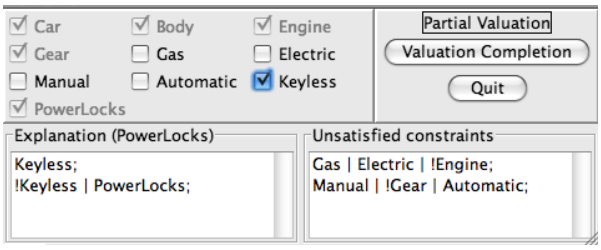

Figure 3. Configuration in action

The program is written entirely in Java and a SAT solver was written as well. Even though a third-party SAT solver could have been used, the author felt as he needs to write one to better understand what he's doing. Moreover, the proof generation is harder to access in other solvers as they need to account for enormous proofs and hence typically record them on the disk.

Nevertheless, it wouldn't be difficult to adapt an existing SAT solver for the same purpose; MINISAT, for instance, is well documented, open-source, and ranks well in competitions [5].

5. Summary and Discussion

The article presents how to use a SAT solver to implement an interactive and backtrack-free tool support for configuration process. Compared to using BDDs, this approach is *lazy*, so to say. A BDD is a pre-compiled representation of the configuration space and it is easy to find small Boolean formulas that explode the BDD in size. On the other hand, to heavily burden a modern SAT solver with a small formula is quite difficult.

Hence, the author believes that the SAT solver approach has a better chance of scaling.

A disadvantage of SAT solvers is that they require a specific input format — typically CNF. That can be overcome by *clausifying* the input, which can be done in such way that the output is linear w.r.t. input. However, such conversion would complicate the explanation generation. For BDDs, this is even worse as the formula's structure is lost.

6. Future Work

Efficiency The algorithm presented in Section 3 is not exchanging information between different calls for different user selections as much as it could have. It could be easily improved by looking at the proofs of locked variables. As long as the user does not alter the decisions on which a proof depends, the pertaining variable will remain locked.

Generalization It is not hard to see how the algorithm TEST-VARS could be generalized for variables with other than just two-valued domains. The algorithm would iterate over each domain, testing each value and eventually analyzing the cardinality of the reduced domain. It is clear, however, that for large domains this would be computationally infeasible and hence a more sophisticated technique is required for such. Most likely, calls to the solver would query for multiple values at a time.

7. Acknowledgments

This work is partially supported by Science Foundation Ireland under grant no. 03/CE2/I303_1.

[3] http://kind.ucd.ie/products/opensource/Config/releases/

References

[1] D. Batory. Feature models, grammars, and propositional formulas. In H. Obbink and K. Pohl, editors, *SPLC '05*, LNCS. Springer-Verlag, 2005.

[2] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In P. Oscar and J. a. Falcão e Cunha, editors, *Proceedings of 17th International Conference on Advanced Information Systems Engineering (CAiSE 05)*, volume 3520 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 1986.

[4] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison–Wesley Publishing Company, 2002.

[5] N. Eén and N. Sörensen. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT '03)*. Springer-Verlag, 2003. Available at http://www.een.se/niklas.

[6] M. Janota and J. Kiniry. Reasoning about feature models in higher-order logic. In P. Kellenberger, editor, *Proceedings of the 11th International Software Product Line Conference, SPLC '07*. IEEE Computer Society, 2007.

[7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA), feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University, Nov. 1990.

[8] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. CHAFF: Engineering an efficient SAT solver. In *39th Design Automation Conference (DAC '01)*, 2001.

[9] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Proceeding of 14th IEEE International Requirements Engineering Conference (RE)*. IEEE Computer Society, 2006.

[10] S. Subbarayan. Integrating CSP decomposition techniques and BDDs for compiling configuration problems. In *Proceedings of the CP-AI-OR*. Springer-Verlag, 2005.

[11] S. Thiel and A. Hein. Modeling and using product line variability in automotive systems. *IEEE Software*, 19(4):66–72, 2002.

[12] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Proceedings of SAT*, 2003.

[13] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2003.