An SMT-based Approach to Automated Configuration (Extended abstract)

Raphaël Michel
CETIC Research Center
Belgium

Arnaud Hubaux
PReCISE Research Center
University of Namur
Belgium

Vijay Ganesh
Massachusetts Institute
of Technology
USA

Patrick Heymans
PReCISE Research Center
University of Namur
Belgium

Abstract

In this paper, we explore a novel application domain for SMT solvers: configuration problems. Configuration problems are everywhere and particularly in product lines, where different yet similar products (e.g., cars or customizable software) are built from a shared set of assets. Designing a product line requires enumerating the different product features and the constraints that determine their valid combinations. Various categories of constraint solvers exist, but SMT solvers appear to be a solution of choice for configuration problems. This is mostly due to their high efficiency and expressiveness which have already proved useful in a variety of practical applications. In this paper, we recall what configuration problems are, describe a language to represent them, and map from configuration problems to the satisfiability problem over SMT theories.

1 Variability Modelling and Configuration Problem

Product lines have become an integral part of modern industry, be it in the manufacturing, the service or the software sectors [20]. Many kinds of products such as cars, insurance contracts or operating systems are developed as assemblies and customizations of reusable components, called *configurations*. Some configurations are valid, some are not, depending on numerous constraints including physical, mechanical, business, legal and design constraints. Moreover, manufacturers have an economic incentive, often dubbed with a legal obligation, in making sure that every possible configuration will result in a viable, safe, secure and useable product. Languages exist in which configuration problems can be encoded. These are often called *variability modeling* languages. Software tools have been built to automate all sorts of reasoning on variability models so as to uncover errors and inconsistencies [2].

Cars are probably the most well-known example of a configurable product. Cars can be equipped with different types of engines, transmissions and gearboxes. Similarly, different bodyworks (sedan, break, sport, cabriolet, etc.), paints, wheel types, and dozens of other options are available for most recent models. When designing a new car model, the manufacturer has to establish a list the various possible options and all the applicable configuration constraints. Some options are obviously incompatible (e.g. a cabriolet cannot have a moonroof), but dependencies among certain options can be subtle (e.g. an integrated GPS requires a color screen, which in turn requires a multifunction steering wheel).

To build a car configurator similar to those found on the websites of car manufacturers, all these constraints need to be taken into account in order to guarantee that the final configuration

will correspond to a car (1) that can actually be built and (2) that the manufacturer wishes to sell. When the variability model of a new car has been produced, the manufacturer can use it to verify properties of the products that can be derived, e.g., "Do all LPG cars also include a regular gas tank?", "Is there any equipment presented to the buyer as an option that is in fact available in all models by default anyways?", etc.

Configurators are *interactive* software tools that record user choices and compute their implications automatically, e.g. update the price of the car, or automatically select / disable related options. For example, if the user selects 'cabriolet', the moonroof option should be either hidden or grayed out. Some configurators even provide a short explanation telling why the option is unavailable. All this needs to be done efficiently in order not to annoy users.

The aforementioned configurator features are often referred to as *choice propagation* and *conflict explanation*. Systematic solutions based on constraint solvers exist to handle these problems. However, constraint handling in configurators is still often developed in ad-hoc manner, i.e. without the help of a solver. This is time-consuming and error-prone. Moreover, maintaining ad-hoc configurators is difficult as the constraints evolve and these are often scattered in many different parts of the code.

Variability modelling languages can be seen as high-level constraint programming languages that address this challenge. They allow one to describe components and constraints in a declarative and natural way. Variability models are then handled by provably reliable constraint solvers to check that the requested configurations are valid. Over the years, researchers have come up with different approaches to that problem using different types of solvers (SAT, BDD, CSP, etc.) and techniques which we describe in the next section. To the best of our knowledge, SMT solvers have never been used for interactive configuration.

Most configuration problems are decisional, i.e., users have to decide whether a given option should be selected or not. However, sometimes configuration problems also involve objective functions for optimizing a certain feature or attribute, e.g., minimize the memory footprint of an operating system running on a mobile device. This paper only sets out to build the foundation for SMT-based decisional configuration and leaves the possible extensions thereof (e.g., SMT with optimization [19]) for future work.

2 Motivation for an SMT-based Approach

Over the years, different technologies have been studied to reason about configuration problems.

The simplest forms of decisional configuration problems are defined as sets of binary decisions governed by basic constraints such as implication and exclusion. Given their nature, these problems are usually modelled and encoded in plain propositional logic. As a result, their verification and the propagation of decisions during interactive configuration can be very efficiently handled by modern SAT solvers [3].

However, casting configuration problems into purely Boolean formulas is often too restrictive. Many problems actually impose constraints over integer, real or string types. Using the higher expressiveness of constraint satisfaction problem solvers (CSPs) [22] is a very common way to address this issue. Standard CSP solvers allow to use several variable types, including Boolean and numerical variables, but some also implement string constraints using regular expressions [12].

Moreover, configuration problems often include variables which can either be activated or deactivated depending on the value of other variables. This need led to the use of Conditional CSPs [7]. Another extension to CSPs, called Generative CSP (GCSP) [7] handles the problem differently by generating variables on the fly during the search process, allowing infinite

configurations to be constructed [1].

Gebser et al. [10] and Friedrich et al. [8] proposed the use Answer Set Programming (ASP), a technology very similar to SAT, based on the stable model semantics of logic programming. Several ASP solvers are based on SAT solvers (e.g. ASSAT [15], SModels-cc [18], and Sup) and others like Clasp and CModels use well-known SAT solving techniques (e.g. Incremental SAT, nogoods, conflict driven). Despite their Boolean nature, their convenient prolog-like syntax allows one to easily handle cardinalities (e.g.: 1a,b,c2 means that at least one and at most two of the three variables must be true). Some ASP solvers like DLV and Clasp [11] also allow one to perform optimizations (#minimize, #maximize).

Other techniques based on knowledge compilation have also been used to handle configuration problems effectively by trading space for performance. Binary Decision Diagrams (BDDs) are one such space efficient data-structure for handling Boolean models. BDDs can be efficient in certain cases where SAT solvers are not. However, beyond a certain threshold the size of the data structure produced during the compilation phase makes this approach intractable for large problems [16]. DNNF, much like BDD is a form of knowledge compilation [6]. BDDs are a special case of DNNF. In their work on enumerating partial configurations, Voronov and al. also consider DNNF algorithms [23].

Dominant approaches to automated configuration are based on SAT, BDD and CSP solvers [22]. In [21], Pohl et al. carried out a performance comparison of such solvers. They selected three different solvers per category. Each solver had to complete a given set of operations on 90 feature diagrams, the most common type of variability models. The largest model contained 287 options. All these models were encoded in propositional logic. Without surprise, SAT solvers performed better for simple validation tasks whereas for more demanding operations such as product enumeration, differences are less clear. BDD solvers typically tend to perform better on larger examples, for instance. Yet, for much larger models (2000+ options), Mendonça et al. [17] show that BDD solvers suffer from intractability while SAT solvers manage to successfully complete a satisfiability check. Regarding expressiveness, SAT is limited due to its Boolean-only nature. CSP-based approaches offer more expressiveness and support various data types, but they are usually less efficient.

Since SMT solvers are often SAT-based, our working hypothesis is that an SMT-based approach will retain the efficiency of modern SAT solvers while being much more expressive. Xiong et al. [24], for instance, have already demonstrated the applicability and high efficiency of SMT for conflict resolution in software configuration. Besides performance, the authors chose to use an SMT solver for its higher expressiveness. Indeed, to apply their approach to operating system configuration, support for arithmetic, inequality, and string operators was required. In [4], Bofill and Palah compare CSP and SMT solvers for a variety of practical applications and show that most of the time SMT solvers are faster. Hence, we propose a novel configuration approach whose reasoning engine builds upon an SMT solver.

3 Modeling a Configuration Problem: The Audi Example

Software engineers rarely encode configuration options and their constraints in a solver-processable format such as SMT-Lib. They rather use higher-level languages like variability models, which are then translated into some solver format. Before delving into this translation, let us first introduce the basic building blocks of a variability model on a concrete example: The Audi car configurator.

Audi is a German car manufacturer. Nowadays, Audi offers 12 different model lines, each available in different body styles with different engines and equipment. We present in Listing 1 a

variability model that contains a sample of all the options available in the Audi car configurator.[1] This model is written in TVL [5], a text-based feature modeling language. TVL may be seen as the last generation of feature diagrams (FD), a notation first introduced in [13]. FDs are now the *de facto* standard variability modelling language. They are usually depicted as trees and capture the commonalities and differences among products in a product family.

Listing 1: Sample TVL model of the Audi car configurator.

```
1  root AudiCar { group allof{ ModelLine, BodyStyle, Engine, Exterior, Model} }
2
3  ModelLine {
4    group oneof {
5      AudiA4 { AudiA4 -> (A4Saloon || A4Avant || S4Saloon || S4Avant) },
6      AudiA6 { AudiA6 -> (A6Saloon || A6Avant) }
7  }
8
9  BodyStyle {
10   group oneof {
11     Saloon { Saloon -> (A4Saloon || S4Saloon || A6Saloon) },
12     Avant { Avant -> (A4Avant || S4Avant || A6Avant) }
13 }
14 Engine{ group allof{ WheelDrive, DriveTrain} }
15
16 WheelDrive{ group oneof{ Quatro, FrontWheelDrive, RearWheelDrive } }
17
18 DriveTrain{ group oneof{ Automatic, Manual, STronic } }
19
20 Gas{ group oneof{ Diesel, Petrol } }
21
22 Exterior{
23   group allof{
24     Color,
25     Wheel,
26     opt ExteriorPackage group someof{ BlackStyling, ContrastRoof }
27   }
28 }
29
30 Color{ group oneof{ Metallic, PearlEffect, Other } }
31
32 Wheel{
33   group allof{
34     Size { enum size in {s15, s16, s17, s18, s19, s20} },
35     TypeWheel
36   }
37 }
38
39 TypeWheel{ group oneof{ Spoke5, Spoke6, Harm7, Hole6} }
40
41 Model{ group oneof{A4Saloon, A4Avant, S4Saloon, S4Avant, A6Saloon, A6Avant} }
```

In Listing 1, the root of the FD is labelled *AudiCar* and has four child features: *ModelLine*, *BodyStyle*, *Engine*, and *Exterior* (line 1). The selection of these child features is determined by the operator **allof** (*and*-decomposition), which means that all the features must be selected; they are thus *mandatory*. Each of these features is then further refined in the rest of the model.

The *ModelLine* (line 3), for instance, has two child features (*AudiA4* and *AudiA6*) whose selection is determined by the **oneof** operator (*xor*-decomposition) at line 4. This operator means that one and only one of the child features can be selected. Each of its child features further constrains the selection of possible options. For instance, the selection of the *AudiA6* implies the selection of the *A6Saloon* or *A6Avant* bodystyle (line 6). Such constraints are called *crosscutting constraints*.

The *Exterior* feature (line 22) introduces two new concepts. First, its child feature *ExteriorPackage* is preceded by the keyword **opt** which means that the feature is *optional*. The

[1] http://configurator.audi.co.uk/

children of this feature are determined by the **someof** operator (*or*-decomposition), which means that at least one option must be selected. TVL also supports generic cardinalities (not illustrated here) of the form $[i..j]$, where i is the minimum number of selectable features and j the maximum.

Finally, the *Size* feature of *Wheel* (line 34) adds another important concept: the *attribute*. In this example, the type of the attribute *size* is an enumeration of values which denotes the possible wheel sizes. In other words, the value of *size* must be one of those listed in the enumeration. Note that this is only one possible type of attributes. An attribute can also be a Boolean, an integer or a string. Furthermore, constraints over these attributes can be defined and include arithmetic (e.g., + or −), comparison (e.g., $\leq$ or $\geq$), and aggregation (e.g., *min* or *max*) operators.

Armed with this intuitive understanding of FDs, we are now set to formally define their fundamentals and their translation into an SMT problem.

4 Feature Diagram Language

TVL provides a concrete syntax for the FD language we use. However, to properly define the translation of TVL into a solver format, we need a mathematical definition of the underlying concepts, irrespective of the concrete representation. In this section, we thus recall the essence of the abstract syntax of TVL from [5]. It will be useful to understand the translation presented in the next section. For brevity, here we do not present the formal semantics but the interested reader can refer to [5]

From an abstract point of view, any FD can be seen as a tree of features, containing a single root, and where each feature is decomposed in one or more features, except for the leaves. Features can be labeled as optional and cardinalities can be used to define the decomposition type of a feature. Features can also have attributes. Attributes of features can be of different types: Boolean, numerical or "enum" (one value among a set of values). As we said previously, FDs are trees, in which features can have "children" features, and a "parent" feature (except for the root). In this context, a "justification rule" stipulates that if a feature is included in the product, then its parent feature must also be included. Then the translation will include the following implication : $B \Rightarrow A$

Definition 1 (Syntactic domain $\mathcal{L}_{FD}$ (Adapted from [5]))**.**

A TVL model $d \in \mathcal{L}_{FD}$ is a tuple $(F, r, \omega, DE, \lambda, A, \rho, \tau, V, \iota, \Phi)$ such that:

- F *is the (non empty) set of features (nodes).*
- $r \in F$ *is the root.*
- $\omega : F \to \{0, 1\}$ *labels optional features with a* 0*.*
- $DE \subseteq F \times F$ *is the decomposition relation between features which forms a tree. For convenience, we will use* $children(f)$ *to denote* $\{g \mid (f, g) \in DE\}$*, the set of all direct sub-features of* f*, and write* $n \to n'$ *sometimes instead of* $(n, n') \in DE$*.*
- $\lambda : F \to \mathbb{N} \times \mathbb{N}$ *indicates the decomposition type of a feature, represented as a cardinality* $[i..j]$ *where* i *indicates the minimum number of children required in a product and* j *the maximum. Note that* and-*,* or-*, and* xor-*decompositions are particular cases of cardinalities. For* $f \in N$*, they are respectively represented by* $[n..n]$*,* $[1..n]$*, and* $[1..1]$*, where* $n = |children(f)|$*.*

- *A is the set of attributes.*
- *$\rho : A \to F$ is a total function that gives the feature declaring the attribute.*
- *$\tau : A \to \{int, enum, bool\}$ assigns a type to each attribute.*
- *V is the set of possible values for enumerated attributes.*
- *$\iota : \{a \in A | \tau(a) = enum\} \to \mathcal{P}(V)$ defines the domain of each enum.*
- *Φ is a formula that captures crosscutting constraints. Without loss of generality, we consider Φ to be a conjunction of formulae on features and attributes.*

Furthermore, each $d \in \mathcal{L}_{FD}$ must satisfy the following well-formedness rules:

- *r is the root: $\forall f \in F(\nexists f' \in F \bullet f' \to f) \Leftrightarrow f = r$,*
- *DE is acyclic: $\nexists f_1, .., f_k \in F \bullet f_1 \to .. \to f_k \to f_1$,*
- *Leaves are $\langle 0..0 \rangle$-decomposed.*
- *Except for the root, each node has a single parent: $\forall f \in F \setminus r : \exists! f' \in F \bullet f' \to f$*

Constraints found in FDs are Boolean expressions over features and attributes that must remain true. These expressions include common Boolean operators (and, or, not, xor, implies). Common arithmetic operators are also available for numerical attributes ("+"", "-", "*", "/", "abs", "$\leq$", "$\geq$", "<", ">", "==", "!="). Comparison operations yield Boolean results that can be combined using the Boolean operators. These expressions are rather straightforward to translate to their equivalent in the solver's input language.

In abstract syntax form, the Audi example from Listing 1 becomes:

$F = \{AudiCar, ModelLine, BodyStyle, Engine, Exterior, \dots\}$
$r = AudiCar$
$\omega(AudiCar) = 1, \omega(ModelLine) = 1, \dots, \omega(ExteriorPackage) = 0, \dots$
$(AudiCar, ModelLine) \in DE, (AudiCar, BodyStyle) \in DE, \dots, (ModelLine, AudiA4) \in DE, \dots$
$\lambda(AudiCar) = [4..4], \lambda(ModelLine) = [1..1], \dots, \lambda(ExteriorPackage) = [1..2], \dots$
$A = \{size\}$
$\rho(size) = Size$
$\tau(size) = enum$
$V = \{s15, s16, s17, s18, s19, s20\}$
$\iota(size) = \{s15, s16, s17, s18, s19, s20\}$
$\Phi = AudiA4 \Rightarrow (A4Saloon \vee A4Avant \vee S4Saloon \vee S4Avant) \wedge$
$\quad AudiA6 \Rightarrow (A6Saloon \vee A6Avant) \wedge$
$\quad Saloon \Rightarrow (A4Saloon \vee S4Saloon \vee A6Saloon) \wedge$
$\quad Avant \Rightarrow (A4Avant \vee S4Avant \vee A6Avant)$

5 Details of the translation from FD to STP

Features, attributes, and constraints over them are the core of an FD. In this section, we present how these constructs are translated into the STP SMT solver's input language [9]. STP is a state-of-the-art SMT solver for a quantifier-free theory of bit-vectors and arrays. The translation from constructs of the FD language (see Definition 1) to STP is described in Table 1.

F, the set of all features defined in the FD is encoded into an array of one-bit bitvectors, where each entry (or index) into the array represents a feature. If the bitvector corresponding

Table 1: Mapping from FDs to STP

FD construct	STP Equivalent	Comment
F : set of features	F: ARRAY BITVECTOR(N) OF BITVECTOR(1)	N = $log(\|F\|)$ Every feature in F is an entry in the array F
$r \in F$	r: BITVECTOR(N)	Root feature r of F indexes the zeroth position of array F
$\omega : \{i \| i \in F\}$	ASSERT(F[i] = 0bin1)	The i^{th} feature in F is not optional
$(f, g) \in F$	f,g : BITVECTOR(N)	Features f, g are converted into indices into array F
For every pair $(f, g) \in DE$	ASSERT((F[g] = 0bin1) $\Rightarrow$ (F[f] = 0bin1))	f is a parent of feature g
$(f, lo, hi) \in \lambda$ means $\forall i \in [1, k] : (f, g_i) \in DE$	ASSERT($lo \leq F[g_1] + F[g_2] + ... + F[g_k] \leq hi$)	The cardinality of the set of features g_i is between lo and hi
for every attribute a of type int over finite range	a : BITVECTOR(N)	Attributes are bitvectors
Arithmetic and logic operations over attributes	Arithmetic and logic operations over corresponding bitvectors	Translation is straightforward
For each pair (feature,attribute)	M: ARRAY BITVECTOR(L) OF BITVECTOR(1) ASSERT(M[i] = 0bin1)	Feature has attribute i
Constraints over features and attributes	Constraints over corresponding bitvectors and arrays	Translation is straightforward

to a feature equals 1, then the feature is included in the FD. Only $log(|F|)$ bits are required

to address this array since any index can be encoded in binary notation. The index of the root feature, named r, is kept in a separate bitvector, of size $log(|F|)$, the same size as any other index. Optional and mandatory features are addressed through the set ω. ω contains all mandatory features, those that are not optional. For each of these features, we force the corresponding bitvector in the array F to be equal to 1, meaning that the feature is included in the product.

FD's are represented as trees. Features have a parent feature (except for the root) and can have children features. Each feature is represented by its index into the F array. Concretely, features are converted to bitvectors of size $log(|F|)$. When a feature is included, its parent must be included as well. This constraint is translated by a group of "assert" statements, each of which handles a pair of features (f, g) where f is the parent of g. If g is included in the product then f must be included as well, meaning that if the g^{th} element of F equals 1, then the f^{th} element must also be equal to 1.

The set of children of a feature form a group, with which a cardinality can be associated, indicating the number of features in F that can be selected. This cardinality is the sum of the corresponding bitvectors in F and must be between the bounds of the cardinality. This is captured by an inequality as shown in Table 1.

Attributes of features are encoded as bit-vector variables whose size depends on the type of the attribute. All the common arithmetic and logic operations on attributes have a straightforward translation to their corresponding operation on bitvectors. Attributes and features are connected using arrays indexed by bit-vectors of suitable length and whose entries are one bit bit-vectors. Each such array corresponds to a feature, and each index in the array corresponds to an attribute. If the i^{th} bitvector of an array equals to 1, then the corresponding feature has the i^{th} value of the corresponding attribute. Note that we assume here that all attributes can be linearly ordered. Finally, additional FD constraints over features and attributes are usually (in)-equality comparisons indicating presence or absence of an attribute or feature. Additionally there can be numeric constraints. All such constraints have a straightforward translation to STP constraints over the corresponding bitvectors and arrays.

6 Preliminary Results

In this section we provide some preliminary results of the STP-based interactive configuration tool that we are developing.

6.1 Benchmarks

We developed the following benchmarks to evaluate our hypothesis: a configuration problem for the Audi car product line and another for the Skoda car product line. The Audi TVL sample was built from their online car configurator[2]. It contains the most common options available including engines, wheels, infotainment, navigation, etc. We also added constraints among these options such as those described in Section 1. We did a similar exercise with the Skoda car configurator[3]. The TVL models of the Audi and Skoda car configurators respectively contain 154 and 253 features with 67 and 179 constraints. In both cases cases, most of these constraints (135 for the Audi example and all for the Skoda example) were expressed in plain propositional logic.

[2] http://configurator.audi.co.uk/
[3] http://www.skoda.co.uk/

6.2 Experimental Setup

All experiments were performed on a MacBook Pro 2.4 GHz, 64 bit and 4 GBytes of RAM. The translation from our FD language to STP is written in Java. We compared performance results between a SAT (CryptoMiniSAT [14]) and an SMT solver (STP). The TVL models were first translated to their equivalent form in STP's input language. Then we compared the performance of the SMT solver using the command `stp -t` (`-t` tells stp to output the execution time) against the performances of CryptoMiniSAT, STP's default backend SAT solver, using `stp -t -r -a -w`. These switches on the command line disable the optimisations and simplifications performed by STP before running CryptoMiniSAT.

6.3 Results and Discussion

STP is approximately twice as fast as pure SAT on our benchmarks. We note that these examples are relatively simple, and as we move to more complex applications we expect SMT solvers to outperform SAT by a significant margin. We are in the process of translating many other configuration problems into SMT problems that are far more complex. For these small models which are mostly Boolean, we observe no significant difference (no more than 2X) between the execution times of STP with and without SMT optimizations. For Boolean models, using an SMT solver offers more expressiveness without additional performance cost.

7 Conclusions and Future Work

In this paper, we proposed a new application for SMT solvers: the configuration problem. While the problem of configuration has already been studied for years using SAT and CSP solvers, each of these solvers have certain limitations which motivate our work. In particular, while SAT solvers are very efficient, their input language is not expressive enough for many kinds of configuration problems. On the other hand, CSP solvers have expressive input languages but are not as efficient as SAT solvers.

SMT solvers, a new breed of solvers, are efficient and expressive for many practical application including configuration problems and hence can be a viable alternative to SAT and CSP solvers. What we reported here is preliminary work on using SMT solvers for solving configuration problems. Specifically, we present a mapping from TVL, a modern variability modelling language, to STP, our SMT solver of choice. Future work will include: (1) Describing how we interact with an SMT solver to perform other interactive configuration tasks such as propagating user choices or explaining conflicts and providing alternatives to solve them; (2) Validating our claim that SMT solvers are more efficient than other types of solvers with similar expressiveness.

References

[1] M. Aschinger, C. Drescher, and G. Gottlob. Introducing loco, a logic for configuration problems. In *Proceedings of the 2nd Workshop on Logics for Component Configuration (LoCoCo'11)*, pages 36–45, Perugia, Italy, 2011.

[2] D. Benavides, S. Segura, and A. Ruiz-Cortes. Automated analysis of feature models 20 years later: A literature reviews. *Information Systems*, 35(6), 2010.

[3] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[4] M. Bofill, M. Palahí, J. Suy, and M. Villaret. SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format. In *Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation (ModRef'09)*, pages 30–44, Lisbon, Portugal, 2009.

[5] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling : Syntax and semantics of TVL. *Science of Computer Programming*, 76:1130–1143, 2011.

[6] A. Darwiche. Compiling knowledge into decomposable negation normal form. In *Proceedings of the 16th international joint conference on Artifical intelligence - Volume 1 (IJCAI'99)*, pages 284–289, Stockholm, Sweden, 1999. Morgan Kaufmann Publishers Inc.

[7] G. Fleischanderl, G. E. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems*, 13:59–68, July 1998.

[8] G. Friedrich, A. A. Falkner, A. Haselböck, G. Schenner, H. Schreiner, and S. A. G. Österreich. (Re) configuration using Answer Set Programming. In *Proceedings of the 12th Workshop on Configuration (ConfWS'11)*, pages 26–35, Barcelona, Spain, 2011.

[9] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th international conference on Computer Aided Verification (CAV'07)*, pages 519–531, Berlin, Germany, 2007. Springer-Verlag.

[10] M. Gebser, R. Kaminski, and T. Schaub. aspcud: A Linux Package Configuration Tool Based on Answer Set Programming. In *Proceedings of the 2nd Workshop on Logics for Component Configuration (LoCoCo'11)*, pages 12–25, Perugia, Italy, 2011.

[11] M. Gebser, B. Kaufmann, and T. Schaub. The conflict-driven answer set solver clasp: Progress report. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, pages 509–514, Potsdam, Germany, 2009. Springer-Verlag.

[12] K. Golden and W. Pang. Constraint reasoning over strings. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03)*, pages 377–391, Kinsale, County Cork, Ireland, 2003. Springer-Verlag.

[13] K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[14] O. Kullmann, editor. *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*. Springer, 2009.

[15] F. Lin and Y. Zhao. Assat: computing answer sets of a logic program by sat solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.

[16] M. Mendonca and A. Wasowski. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240, San Francisco, USA, 2009. Carnegie Mellon University.

[17] M. Mendonça. *Efficient reasoning techniques for large scale feature models*. PhD thesis, 2009.

[18] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A system for answer set programming. *CoRR*, cs.AI/0003033, 2000.

[19] R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. In A. Biere and C. P. Gomes, editors, *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 156–169. Springer, 2006.

[20] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[21] R. Pohl, K. Lauenroth, and K. Pohl. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, ASE '11, pages 313–322, Washington, DC, USA, 2011. IEEE Computer Society.

[22] F. Rossi, P. v. Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[23] A. Voronov, K. Å kesson, and F. Ekstedt. Enumeration of valid partial configurations. In *Proceedings of the 12th Workshop on Configuration (ConfWS'11)*, pages 25–31, Barcelona, Spain, 2011.

[24] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, Zurich, Switzerland, 2012. IEEE Computer Society. (To appear).