Conflict Handling in Product Configuration using Answer Set Programming

Konstantin Herud[1], Joachim Baumeister[1,2], Orkunt Sabuncu[3,4] and Torsten Schaub[4,5]

[1]*denkbares GmbH, Germany*
[2]*University of Würzburg, Germany*
[3]*TED University, Turkey*
[4]*Potassco Solutions, Germany*
[5]*University of Potsdam, Germany*

Abstract
Product configuration is one of the most important industrial applications of artificial intelligence. In order to enable customers to individualize complex products, usually logical configuration models are necessary. One challenge that exists here is the communication of product knowledge to the customer. This work deals with the situation of invalid customer requirements under given logical constraints. The goal is to explain *why* configurations are invalid and *how* they can be minimally changed to become valid again. This involves computing all minimal subsets of both constraints and requirements that make the configuration unsatisfiable. For this, we present a novel approach based on the declarative paradigm of Answer Set Programming. We empirically demonstrate its suitability for real-time configurators with thousands of features and constraints. The approach thus fills the gap of an easy-to-implement as well as high-performing conflict resolution component.

Keywords
Product Configuration, Conflict Explanation, Conflict Resolution, Answer Set Programming, Minimally Unsatisfiable Subsets

1. Introduction

Product configuration deals with the commercial interest of making an offering more attractive by allowing it to be personalized by customers. As a result, products or services are more likely to be purchased. Instead of offering a predefined set of cars, for example, customers can instead assemble the exact car they want from components such as different wheels, colors, and assistants. Many products are also often too complex that there is no other option than to configure them specifically for each customer. This typically happens with non-standardizable parameters, such as the dimensions and quantity of window blinds. Similar to software development, product configuration has thus accompanied many companies for a long time. It is therefore considered one of the most successful applications of artificial intelligence. Product configuration is not only used for simple products, but for a variety of diverse problems [1]. Examples include railway interlocking systems [2], cement plants [3], mobile phone networks

Federated Logic Conference: ICLP Workshops, July 31 – August 12, 2022, Haifa, Israel
konstantin.herud@denkbares.com (K. Herud); joachim.baumeister@denkbares.com (J. Baumeister); orkunt.sabuncu@tedu.edu.tr (O. Sabuncu); torsten@cs.uni-potsdam.de (T. Schaub)

[4], offers, contracts, user manuals, or technical documentation [5], and services like elevator maintenance [6]. Despite its long history and profitability, there are still several issues that make implementing product configuration a challenge. Here, there are two sides: On the one hand, the business side, where the main challenge is collaboratively developing and continuously maintaining a knowledge base about a product. On the other hand, the customer side, which is concerned with intuitively communicating the knowledge. Both sides divide into numerous sub-challenges. This paper addresses one challenge, namely resolving conflicts interactively with the customer during the configuration process. A conflict here denotes an unsatisfiable set of user selected feature requirements which result in a configuration that cannot be manufactured or sold. While this notion will be defined in more detail, we pursue two goals based on it in this work:

1. **Explanation**: It should be explained how the user requirements violate the constraints of the configuration model.
2. **Resolution**: Recommendations should be made to the user as to which requirements they should change to resolve the conflict.

To avoid overwhelming the user with unnecessary information, minimal subsets have to be calculated in both cases. For Goal 1 these are the subsets of unsatisfied constraints. For Goal 2 they are the subsets of feature selections that make the configuration unsatisfiable. Although various solutions for computing minimal subsets exist, e. g., *QuickXPlain* [7], these encounter several problems. First, it is costly and error-prone to implement these solutions by oneself. QuickXPlain is one of the most popular approaches, yet there is no reference implementation. Second, regarding performance, there are both qualitative and quantitative challenges. Since configurators are mostly interactive, response times must be in real time and scalable to a large number of customers. To address all of these challenges, we present a novel approach based on Answer Set Programming (ASP), a declarative problem solving paradigm [8, 9]. The contribution of this work therefore is threefold:

- We present a *general* and *extensible* framework for modeling product configuration using ASP.
- We present an *easy-to-implement* approach for explaining and resolving conflicts during product configuration.
- We demonstrate the *high performance* of the approach, which makes it well suited for real-time applications.

For this, in Section 2, we first define product configuration and in particular the explanation as well as the resolution of conflicts. Based on this definition of the problem, we show the related literature in Section 3. Here, we first discuss the broader context, then specifically ASP and the algorithm needed for our implementation. Subsequently, we present this implementation in Section 4 and show an evaluation of our approach in Section 5. Finally, we conclude this work in Section 6.

2. Definition

Product configuration is a well researched area and there are several efforts for a general ontology [10, 11, 12]. However, we keep the definition of product configuration in Section 2.1 as simple as possible for the scope of this work. The ideas presented are nevertheless easily extendable to more complex ontologies. Moreover, the approaches can also be applied to the superset of configuration problems in general. Section 2.2 will therefore formally define and illustrate the problem of explaining conflicts during (product) configuration. Section 2.3 follows equivalently with the resolution of conflicts.

2.1. Product Configuration

Product configuration involves the feature-based personalization of a product, which is assembled from various components. A simple example of this is the configuration of a car. Customizable features here include the type of steering wheel and the color of the casing. The lowest common denominator of general ontologies are therefore *features* and their *type* in the knowledge modeling, i. e., variable characteristics of the product and their domains.

Definition 1 (Feature)**.** The properties of a product are specified by $1 \leq i \leq k$ *features.* A feature f_i is a variable defined by its feature *type.* The type $domain(f_i)$ of a feature determines its discrete, finite, and non-empty domain.

We use typewriter font for concrete examples and write features with a capital initial letter. Domain values are completely capitalized. For example, the color of a car could be described by a feature `Color`, where $domain(\texttt{Color}) = \{\texttt{BLACK}, \texttt{WHITE}, \texttt{RED}\}$. Features define the structural knowledge about configurable aspects in a product. But a large part of the knowledge is behavioral, i. e., the constraints on the interaction between features. The set of all *possible* configurations results from the cross product of the domains of all features. However, usually only a fraction of them form *valid* configurations, i. e., combinations of feature values that can be manufactured. To capture this set of valid solutions in the space of all possible solutions, constraints are defined.

Definition 2 (Constraint)**.** A *constraint* c restricts the solution space of possible configurations. It is defined as a relation that maps a configuration X to a boolean truth value, i. e.,

$$c : X \to \{\top, \bot\}. \tag{1}$$

While the configuration will be defined in more detail shortly, the combination of features and constraints of a product forms a *knowledge base.*

Definition 3 (Configuration Knowledge Base)**.** A *configuration knowledge base* is a triple (F, D, C), where F is the set of all features, D is the set of all feature types, and C is a set of constraints over F.

Although here we bypass the dynamic nature of product configuration, where features become active depending on the configuration, our approach can be easily extended to do so. This requires introducing cardinalities for features, e. g., to define optional features. Regardless, the knowledge base can be used to offer individual *configurations* to customers.

Definition 4 (Configuration)**.** A *configuration* X binds values x to features $f_i \in F$ in a knowledge base (F, D, C).

$$X = \{f_i = x \mid f_i \in F \wedge x \in \mathit{domain}(f_i)\}\,. \tag{2}$$

X is *complete*, if (3) holds, and *valid* if (4) holds.

$$\mathit{complete}(X) \,:\, \forall f_i \in F \,:\, f_i \in X \tag{3}$$

$$\mathit{valid}(X) \,:\, \forall c \in C \,:\, c(X) = \top \tag{4}$$

A configuration is therefore merely the finished process of feature binding where exactly one value exists for each feature. The *user requirements* U are a partial configuration, where each feature assignment $f_i = x$ is explicitly given by the user. Neither $\mathit{complete}(U)$ nor $\mathit{valid}(U)$ have to be true. If $\mathit{valid}(U)$ holds, the configuration task consists of completing U to a configuration X such that $\mathit{complete}(X)$ and $\mathit{valid}(X)$ holds. Note that multiple configurations may be possible for completing a configuration task. The amount of options is upper bounded by the cross product of the domains of all features. There is not always a solution to a configuration problem, though, namely when $\mathit{valid}(U)$ does not hold. In this paper we are interested in exactly this case, which we refer to as *conflict*. To sell a configuration despite invalid requirements, the conflict must be explained and resolved interactively with the customer.

2.2. Explaining Conflicts

Explaining conflicts requires two steps:

1. An analysis of the set of violated constraints
2. A description of the violated constraints in natural language

Even if the general definition 2 of constraints as arbitrary relations is restricted to, for example, first order logic, it is very difficult to automatically generate natural language explanations. While it is easy to assemble explanations from blocks of text for each logical operator, the result is typically hard to parse and not helpful to the user. In most cases, it is also not in a company's best interest to disclose the exact constraints of the knowledge base to the public. Instead, to concisely explain the higher purpose of violated constraints and in particular the interaction of multiple constraints, usually external knowledge is required. We therefore argue that explanations have to be manufactured in natural language as part of knowledge modeling. One solution to this may be the use of modern language modeling and its ability for *zero shot learning* [13]. This involves solving problems at test time that did not occur during training. Here, for example, the models are able to summarize error messages from long log files into natural language explanations. However, we leave this research question open as future work.

Instead, this paper focuses on Step 1 — the analysis of violated constraints. These are especially important for debugging during the development of the knowledge base. Although it is easy to output which constraint was violated during the reasoning process, two challenges arise:

1. A conflict only arises through the interaction of several constraints. Each individual constraint would not yield an unsatisfiable configuration.
2. Only the minimal set of constraints yielding an unsatisfiable configuration has to be determined. Note that there may be several independent sets.

As an example, we look again at the car configuration problem. Here, among others, four constraints exist, for example, as a consequence of a larger table of allowed values.

- c_0 : $\texttt{Body} = \texttt{OFF\_ROAD} \rightarrow \texttt{Drive} = \texttt{ALL\_WHEEL}$
- c_1 : $\texttt{Drive} = \texttt{ALL\_WHEEL} \rightarrow \texttt{Transmission} = \texttt{AUTOMATIC}$
- c_2 : $\texttt{Transmission} = \texttt{AUTOMATIC} \rightarrow \texttt{CruiseControl} = \texttt{TRUE}$
- c_3 : $((\texttt{Leather} = \texttt{TRUE}) \wedge (\texttt{Color} = \texttt{BLACK} \vee \texttt{Color} = \texttt{WHITE})) \vee \ldots$

Note that this example formally hurts the definition of a constraint as a relation, but it is purely for intuition. `Body` determines the type of car, e. g., sport, cabriolet, or SUV, and `Color` its finish. A customer now wants to buy a black off-road vehicle, i. e., $U_0 = \{\texttt{Body} = \texttt{OFF\_ROAD}, \texttt{Color} = \texttt{BLACK}\}$. Since they want to save money, they choose faux leather and deactivate the cruise control assistant resulting in $U_1 = U_0 \cup \{\texttt{Leather} = \texttt{FAUX}, \texttt{CruiseControl} = \texttt{FALSE}\}$. The example is tailored to intuitively see that the user requirements are thereby unsatisfiable. In addition, both Challenges 1 and 2 are illustrated.

1. There is a chain of constraints which only together yield unsatisfiability, namely c_0, c_1, and c_2 together with requirements for `CruiseControl` and `Body`.
2. Two independent conflicts occur. First, the conflict chain of the previous point. Second, the choice of color `Color` which is incompatible with the faux leather `Leather` by c_3.

Based on this, we define the goal of analysing conflicts in Equation 5. It consists of determining the set C^- of all $\subseteq$-minimal sets of constraints which alone lead to the unsatisfiability of user requirements U.

$$C^- = \left\{C_i^- \subseteq C \mid \neg \mathit{valid}(U) \text{ given } \left(F, D, C \cap C_i^-\right)\right\} \\ \text{such that } \forall C_i, C_j \in C^- : C_i \nsubseteq C_j. \tag{5}$$

2.3. Resolving Conflicts

In practice, a suggestion should be offered to the user on how to change the set of unsatisfiable requirements U to resolve all conflicts. For this, a subset of U needs to be found, whose removal makes it possible to complete U into a valid configuration X again. Again, there are two challenges:

1. There can be different ways to resolve a conflict. For example, with a constraint on a combination of multiple features, each individual feature can be reset on its own.
2. Each way to resolve a conflict should be minimal, i. e., no subset of any possibility also leads back to satisfiability.

Given Challenge 1, we are interested in computing all $\subseteq$-minimal candidates for resolving the conflict. These can be subsequently ranked by a recommender system to maximize user or company interest. We again examine the car example from Section 2.2. To resolve the two conflicts under the given four constrains and the requirements

$$\{\texttt{Body} = \texttt{OffRoad}, \texttt{Color} = \texttt{Black}, \texttt{Leather} = \texttt{Faux}, \texttt{CruiseControl} = \texttt{False}\}$$

there are four optimal possibilities for removing requirements.

1. {`CruiseControl`, `Color`}
2. {`CruiseControl`, `Leather`}
3. {`Body`, `Color`}
4. {`Body`, `Leather`}

Similarly to explaining conflicts the task of resolving conflicts defines itself by calculating $\subseteq$-minimal answers. Instead of the constraints, however, all sets U^- of the user requirements are to be computed, by whose exclusion the configuration becomes valid again.

$$\begin{gathered} U^- = \left\{ U_i^- \subseteq U \mid \mathit{valid}\left(U \setminus U_i^-\right) \text{ given } (F, D, C) \right\} \\ \text{such that } \forall U_i, U_j \in U^- : U_i \not\subseteq U_j. \end{gathered} \tag{6}$$

Note that every single set $U_i^- \in U^-$ must contain at least one value for all $C_i^- \in C^-$, that gives rise to the respective conflict, i. e., $|U_i| \geq |C^-|$. For this reason, each of the four conflict resolution options consists of two features, one for each of the conflicts.

3. Background

Based on the preceding problem definition, we now contextualize our work in the literature in Section 3.1 and present the background of Answer Set Programming in Section 3.2.

3.1. Related Work

The explanation of unsatisfiable constraint systems has a large background [14, 15, 16, 17, 18, 19, 20, 21, 22, 23]. The vast majority of these approaches are based on the notion of a *Minimally Unsatisfiable Subset (MUS)* [24]. Given a set of constraints C, a subset $\hat{C} \subseteq C$ is a MUS if $\hat{C}$ is unsatisfiable and for each $\hat{C}_i \subset \hat{C}$ is satisfiable. Note that our goal of explaining conflicts corresponds to the computation of all MUSs. These are computed over the set of constraints in the knowledge base. The set of user requirements U corresponds to hard, non-relaxable constraints. Besides MUSs, another important concept is *Minimal Correction Subsets (MCS)* [24]. Given a set of constraints C, a subset $\hat{C} \subseteq C$ is a MCS if $C \setminus \hat{C}$ is satisfiable and for each $\hat{C}_i \subset \hat{C}$ is unsatisfiable. Similarly, our goal of conflict resolution corresponds to the computation of MCSs. This time U is a set of relaxable soft constraints and the knowledge base constraints are hard. Finally, there is an important relation between MUS and MCS [24]. Let X be a finite collection of sets. A hitting set $HS(X)$ is defined as $\forall X_i \in X : HS(X) \cap X_i \neq \emptyset$. It is *minimal* if there is no proper subset of it that is also a hitting set of X. The *MUS/MCS* duality states that any MCS of a problem instance is a hitting set of all MUSs of that instance and, correspondingly, any MUS is a hitting set of all MCSs. Similar to many other methods, we make use of this relationship.

A recent survey classifies methods along four dimensions [25]: First, whether the solution is solver-specific or solver-agnostic. Solver-agnostic methods typically use a-posteriori analyses to generate explanations. In this respect, our approach is specific to *Conflict Driven Nogood Learning* [26], though agnostic to the specific implementation. The second dimension is user or solver focused. Our explanations are directed at users rather than to support the search process of a solver. The third dimension is about various quality metrics, e. g., to prefer the smallest possible explanation size. Since our methods potentially find all unsatisfiable constraint and user requirement sets, any kind of ranking metrics can be easily implemented. This also provides

an answer to the last dimension, namely whether all or only a subset of the declarations are computed. It is relevant here to mention that our conflict analysis is only partially any-time capable, but our conflict resolution works completely any-time, which is explained in more detail in Section 4.
While there is a lot of literature on general approaches for constraint satisfaction problems, the literature specific to product configuration is much sparser [7, 27]. Lastly, there is a small amount that deals with product configuration using ASP, however we are not aware of any approaches that perform conflict analysis during product configuration using ASP. Our work thus fills this gap of an easy-to-implement conflict resolution component.

3.2. Answer Set Programming

A logic program in *Answer Set Programming* consists of rules of the form

$$\mathtt{a}_1 \ ; \ \dots \ ; \ \mathtt{a}_m \ \mathtt{:-} \ \mathtt{a}_{m+1} \ , \ \dots \ , \ \mathtt{a}_n , \ \mathbf{not} \ \mathtt{a}_{n+1} \ , \ \dots \ , \ \mathbf{not} \ \mathtt{a}_o .$$

where each $\mathtt{a}_i$ is an atom of form $\mathtt{p(t}_1\mathtt{,...,t}_k\mathtt{)}$ and all $\mathtt{t}_i$ are terms, composed of function symbols and variables. For $1 \le m \le n \le o$, atoms $\mathtt{a}_1$ to $\mathtt{a}_m$ are often called head atoms, while $\mathtt{a}_{m+1}$ to $\mathtt{a}_n$ and **not** $\mathtt{a}_{n+1}$ to **not** $\mathtt{a}_o$ are also referred to as positive and negative body literals, respectively. An expression is said to be ground, if it contains no variables. As usual, **not** denotes (default) negation. A rule is called a fact if $m = n = o = 1$, normal if $m = 1$, and an integrity constraint if $m = 0$. In this work, we deal with normal logic programs only. Semantically, a logic program induces a set of stable models, being distinguished models of the program determined by the stable models semantics [28].

To ease the use of ASP in practice, several extensions have been developed. First of all, rules with variables are viewed as shorthands for the set of their ground instances. Further language constructs include conditional literals and cardinality constraints [29]. The former are of the form $\mathtt{a}\mathtt{:}\mathtt{b}_1\mathtt{,...,b}_m$, the latter can be written as $\mathtt{s}\ \{\mathtt{d}_1\mathtt{;...;d}_n\}\ \mathtt{t}$, where $\mathtt{a}$ and $\mathtt{b}_i$ are possibly negated (regular) literals and each $\mathtt{d}_j$ is a conditional literal; $\mathtt{s}$ and $\mathtt{t}$ provide optional lower and upper bounds on the number of satisfied literals in the cardinality constraint. We refer to $\mathtt{b}_1\mathtt{,...,b}_m$ as a condition. The practical value of both constructs becomes apparent when used with variables. For instance, a conditional literal like `a(X):b(X)` in a rule's body expands to the conjunction of all instances of `a(X)` for which the corresponding instance of `b(X)` holds. Similarly, `2 {a(X):b(X)} 4` is true whenever at least two and at most four instances of `a(X)` (subject to `b(X)`) are true. Note that we name a normal rule with a cardinality constraint construct as the head a choice rule.

Next, let us consider a system directive particular to *clingo*. *Clingo* offers means for manipulating the solver's decision heuristics. We rely on this capacity to compute answer sets that are subset minimal with respect to the choices of the solver. Such heuristic directives are of form

$$\mathbf{\#heuristic}\ \mathtt{a}\mathtt{:}\mathtt{b}_1\mathtt{,...,b}_m\mathtt{.}\ \mathtt{[w,m]}$$

where $\mathtt{a}\mathtt{:}\mathtt{b}_1\mathtt{,...,b}_m$ is a conditional literal; `w` is a numeral term and `m` a heuristic modifier, indicating how the solver's heuristic treatment of a should be changed whenever $\mathtt{b}_1\mathtt{,...,b}_m$ holds. The modifier `false`, for instance, not only assigns `w` as the level of a given that *clingo* decides first upon atoms of the highest level, but also enforces that a becomes false whenever it is chosen by the solver.

Finally, Algorithm 1 for finding ⊆-minimal answers involves two ideas [30].

1. First, all atoms T that are to be ⊆-minimized are set to false using heuristic directives before deciding on the truth values of other atoms. This ensures that the first answer S found is ⊆-minimal with respect to T.
2. To enumerate all ⊆-minimal responses, the solver adds a *nogood* over the true atoms in $S \subseteq T$ after each call. This constraint ensures that no superset of the ⊆-minimal answer is found, which ultimately leads to enumerating all ⊆-minimal answers.

The loop of solving and adding nogoods is iterated until the program is no longer satisfiable and thus all answers have been found.

```
Input: Program $P$ and atoms $T$ to subset minimize
foreach $x \in T$ do
    SetSign($x$, $false$, 1);
while satisfiable do
    S←Solve(P);
    P←AddConstraint($\bot \leftarrow a_0, \dots, a_n$ for $\{a_0, \dots, a_n\} = T \cap S$);
    Output($S$);
```

Algorithm 1: Algorithm used for finding minimal subsets [30]

4. Implementation

This section shows how to make use of ASP for the problem definition of Section 2. For that, we first define the product configuration knowledge base in ASP syntax. Then, both conflict analysis and resolution are presented. Note that ASP generally restricts the definition of constraints as arbitrary relations. However, *clingo*offers with so-called theories capabilities to introduce foreign solving techniques and thus to remove this restriction [31]. For the scope of this paper, we nevertheless limit the expressive power of constraints to the capabilities of ASP.

4.1. Problem Modelling

In ASP, problem instances are specified as facts. For this, we first formulate the structural and behavioral knowledge. We then describe the solution path of the search problem, known as *encoding*.

4.1.1. Structural Knowledge

The main predicate of the hierarchy is `feature/2` with arguments TYPE, and NAME. Note that by convention capitalized symbols define variables. Here for feature f_i, TYPE refers to *domain* (f_i) and NAME is the unique feature name. These are specified as facts in the program, i. e., the variables are replaced with specific values. The domain of concrete features is defined using `domain/2` to specify a value VALUE for domain TYPE. Lastly, `value/2` exists to bind values VALUE to features FEATURE. This predicate is used during inference and to specify user requirements.

4.1.2. Behavioral Knowledge

Since constraints are part of a specific problem instance, we reify them into facts from which respective rules are instantiated [31]. For this purpose, we formulate a simplified meta-encoding to instantiate equations, disjunctions, conjunctions, and negations from facts. The facts contain literal identifiers, e. g. variables `X, L, R, O`, or `A` as arguments, which are abstract symbols.

```
holds(X) :- eq(X, L, R), value(L, VL), value(R, VR), VL = VR.
holds(X) :- or(X), #count{ O : or(X, O), holds(O)} > 0.
holds(X) :- and(X), #count{ A : and(X, A), not holds(A)} = 0.
holds(X) :- neg(X, N), not holds(N).
```

Thus, arbitrarily complex syntax trees of logical statements can be created. Note that this encoding can be easily extended, e.g. to formulate inequalities or arithmetic expressions. Wherever we use predicates with the same name but different arity, e. g. `or/1` and `or/2`, the shorter predicates just omit the last arguments.

4.1.3. Generate and Test

We define the encoding of the problem according to the "generate and test" ASP convention. First, a choice rule describes a set of all possible solutions, i. e., a simple superset of the set of solutions to the given search problem.

```
1 {value(F, V) : domain(T, V)} 1 :- feature(T, F).
```

Second, it is followed by an integrity check, which eliminates all solutions that lead to an invalid configuration, i. e.,

```
:- constraint(C), not holds(C).
```

Here `C` points to the root literal of the syntax tree for any constraint.

4.2. Explaining Conflicts

Conflict analysis requires finding the set of all $\subseteq$-minimal constraint sets of a knowledge base (F, D, C) that alone yield unsatisfiability. Instead of directly calculating these minimally unsatisfiable subsets (MUSs) it is easier to calculate minimal correction sets (MCSs). This requires only an additional predicate `mcs/1`, which takes as argument the literal `C` of a `constraint/1` atom. The solver decides about the truth value of `mcs/1` to deactivate constraints as desired. To then determine all MCSs, the solver initially assigns all `mcs/1`-atoms to false, as presented in Section 3.2, and learns nogoods about all included `mcs/1`-atoms after each answer. This requires a modification of the previous integrity check: it cannot be that a constraint does not hold and is not part of a MCS.

```
{mcs(C) : constraint(C)}.
:- constraint(C), not holds(C), not mcs(C).
#heuristic mcs(C) : constraint(C). [1,false]
```

It is then easy to use the MUS/MCS duality from subsection 3.1 to compute the corresponding MUSs, i. e., the hitting sets of the MCSs. A pure solution in ASP without further implementation requires for example the encoding below.

```
{mus(C) : mcs(I, C)}.
:- mcs(I), #count{ C : mus(C), mcs(I, C)} != 1.
```

Here `mcs/2` is the `I`-th MCS with constraint `C` and `mus/1` is a MUS with constraint `C` (the index is implicitly given by the answer set). Note that this happens in a separate step. However, the complexity is much less than computing the MCSs and thus negligible. For the car example in Section 2.2 three MCSs result:

1. $\{c_0, c_3\}$
2. $\{c_1, c_3\}$
3. $\{c_2, c_3\}$

From which the following MUSs are calculated:

1. $\{c_0, c_1, c_2\}$
2. $\{c_3\}$

Besides simplicity, a major advantage is the loose description of constraints. The approach is thus agnostic about their concrete implementation. E. g., theory atoms [32] can be easily implemented to simplify numerical calculations [33].

4.3. Resolving Conflicts

For conflict resolution, we need to find the ⊆-minimal set of user requirements U to change. Similar to the previous section, we introduce two predicates `mcs/2` and `assumption/2`. Both take as arguments a FEATURE and VALUE. Note that this formulation is a simplification and can easily be generalized to arbitrary arguments. We then formulate an encoding around the original choice rule for selecting values out of feature domains. Again, we let the solver decide on the truth value of the `mcs/2` atom. If it chooses false, the passed assumption is true, i. e., the original user requirement. Otherwise, the solver can make a free choice for the feature. To determine the ⊆-minimal required changes, the solver again uses the algorithm from Section 3.2 and initially assigns all `mcs/2`-atoms to false. The following listing shows the required modification of the ASP encoding, with the first line giving an example of `assumption/2`.

```
assumption(color, black).
{mcs(F, V)} :- assumption(F, V).
value(F, V) :- assumption(F, V), not mcs(F, V).
#heuristic mcs(F, V) : assumption(F, V). [1,false]
```

5. Evaluation

Finally, we demonstrate the high performance of our approach in Section 5.2. For legal reasons, we do not publish real knowledge bases, but generate synthetic knowledge bases utilizing real ones in Section 5.1.

5.1. Dataset

To evaluate the performance of our approach, we probabilistically generate synthetic knowledge bases (F, D, C) for publication. Here, the parameters of the probabilistic distributions are extracted from real knowledge bases of industrial partners. First, F and D are generated. The

Table 1
The results of our experiments. Each value is the average of 100 problem instances. *Size* refers to the amount of both features and constraints in a knowledge base. *Choices* refers to the amount of decisions the solver made about choice rules. All experiments were performed with a single CPU core.

(a) Explanation

Size	Grounding	Solving	Choices
10	1.1 ms	0.24 ms	7
100	12.8 ms	2.4 ms	94
1,000	97.4 ms	23.4 ms	954
10,000	3.1 s	1.0 s	27,885

(b) Resolution

Size	Grounding	Solving	Choices
10	1.4 ms	0.4 ms	29
100	18.6 ms	8.0 ms	2,123
1,000	135.4 ms	124.8 ms	54,009
10,000	4.2 s	2.7 s	1,080,180

number of options of each feature follows a log-normal distribution. Second, a predetermined number of constraints are generated according to the scheme *conditions* → *consequences*. The *conditions* are a binomially distributed conjunction of $f_i = x$ or $f_i \neq x$ pairs. Here, for each constituent of the conjunction, an $f_i \in F$ is drawn at random without replacement, and correspondingly an $x \in \mathit{domain}(f_i)$. An empty set of conditions is allowed. The *consequences* represent a set of allowed feature combinations, which is the most common constraint type in our real knowledge bases. We first randomly draw a binomially distributed set of more than one feature. This set represents the columns of the combination table. Then, the number of allowed feature-value combinations, i. e., the rows of the table, is generated log-normally distributed. Each combination here consists of a conjunction over a range of values for each feature. Each constituent of the conjunction, i.e., a cell of the table, in turn describes as a disjunction a set of allowed or disallowed values. We ensure that each generated knowledge base allows at least $\min\left(1,000,000, |F|^2\right)$ valid configurations.

Finally, we generate problem instances, i. e., $U : \neg \mathit{valid}(U)$. For this we draw a log-normally distributed percentage p of constraints to be violated and slightly modify the integrity check of our encoding:

```
:- #count{ C : constraint(C), not holds(C) } != n.
```

Here $n = \mathrm{round}(p * |C|)$. We then let the solver decide randomly during search to generate diverse instances. Our data is publicly available. [1]

5.2. Results

The results of the experiments are shown in Table 1. We evaluate problem instances with 10, 100, 1000 and 10,000 both features as well as constraints. For each size, we create 100 problem instances and report the mean. On average, the programs have relative to their size 890, 10,625, 95,617, and 2,739,913 atoms, and a similar albeit slightly higher number of rules. Note that grounding may typically be performed only once per knowledge base as a preprocessing step. Subsequently, for example, assumptions or multishot capabilities can be used to solve all problem instances [31].

[1]https://github.com/kherud/product-configuration-synthetic-data

6. Conclusion

The explanation of conflicts in unsatisfiable constraint systems has historically been of great interest. Although a variety of approaches exist, they are often purely of academic relevance. Since there are rarely reference implementations, own implementations require profound scientific knowledge. These problems make conflict analysis a major challenge in practice. Product configuration is one of the most important practical areas of artificial intelligence. Yet, comparatively little literature exists that addresses its application in practice. Answer Set Programming (ASP) is the right tool due to its declarative abstraction of the problem description from the solution path. Thus, we first developed a general and extensible framework for modeling product configuration in ASP. We then showed how both the explanation of conflicts and their resolution can be solved in under ten lines of declarative ASP syntax. Finally, our empirical evaluation showed that the technique is well suited for the high performance demands of real-world applications. A major scientific challenge that remains open is communicating the identified conflicts to non-expert users in natural language.

References

[1] A. Felfernig, L. Hotz, C. Bagley, J. Tiihonen, Knowledge-based Configuration: From Research to Business Cases, Morgan Kaufmann, Amsterdam, 2014.

[2] A. Falkner, H. Schreiner, Siemens: Configuration and reconfiguration in industry, Knowledge-Based Configuration: From Research to Business Cases (2014) 199–210. doi:10.1016/B978-0-12-415817-7.00016-5.

[3] K. Orsvärna, M. H. Bennick, Tacton: Use of tacton configurator at flsmidth, in: Knowledge-Based Configuration, Morgan Kaufmann, 2014.

[4] I. Nica, F. Wotawa, R. Ochenbauer, C. Schober, H. F. Hofbauer, S. Boltek, Kapsch: Reconfiguration of mobile phone networks, in: Knowledge-Based Configuration, Morgan Kaufmann, 2014.

[5] R. Rabiser, M. Vierhauser, M. Lehofer, P. Günbacher, T. Männistö, Configuring and generating technical documents, in: Knowledge-Based Configuration, Morgan Kaufmann, 2014.

[6] J. Tiihonen, W. Mayer, M. Stumptner, M. Heiskala, Configuring services and processes, in: Knowledge-Based Configuration, Morgan Kaufmann, 2014.

[7] U. Junker, QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems, in: D. L. McGuinness, G. Ferguson (Eds.), Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA, AAAI Press / The MIT Press, 2004, pp. 167–172. URL: http://www.aaai.org/Library/AAAI/2004/aaai04-027.php.

[8] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Answer Set Solving in Practice, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2012. URL: https://doi.org/10.2200/S00457ED1V01Y201211AIM019. doi:10.2200/S00457ED1V01Y201211AIM019.

[9] V. Lifschitz, Answer Set Programming, Springer, 2019. URL: https://doi.org/10.1007/978-3-030-24658-7. doi:10.1007/978-3-030-24658-7.

[10] S. Mittal, F. Frayman, Towards a generic model of configuraton tasks, in: Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 2, IJCAI'89, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989, p. 1395–1401.

[11] T. Soininen, J. Tiihonen, T. Männistö, R. Sulonen, Towards a general ontology of configuration, Artif. Intell. Eng. Des. Anal. Manuf. 12 (1998) 357–372. URL: http://journals.cambridge.org/action/displayAbstract?aid=38651.

[12] U. Junker, Configuration, in: F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, volume 2 of *Foundations of Artificial Intelligence*, Elsevier, 2006, pp. 837–873. URL: https://doi.org/10.1016/S1574-6526(06)80028-3. doi:10.1016/S1574-6526(06)80028-3.

[13] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners, in: H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin (Eds.), Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[14] R. Bruni, On exact selection of minimally unsatisfiable subformulae, Ann. Math. Artif. Intell. 43 (2005) 35–50. URL: https://doi.org/10.1007/s10472-005-0418-4. doi:10.1007/s10472-005-0418-4.

[15] J. Huang, MUP: a minimal unsatisfiability prover, in: T. Tang (Ed.), Proceedings of the 2005 Conference on Asia South Pacific Design Automation, ASP-DAC 2005, Shanghai, China, January 18-21, 2005, ACM Press, 2005, pp. 432–437. URL: https://doi.org/10.1145/1120725.1120907. doi:10.1145/1120725.1120907.

[16] F. Hemery, C. Lecoutre, L. Sais, F. Boussemart, Extracting mucs from constraint networks, in: G. Brewka, S. Coradeschi, A. Perini, P. Traverso (Eds.), ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings, volume 141 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2006, pp. 113–117. URL: http://www.booksonline.iospress.nl/Content/View.aspx?piid=1657.

[17] É. Grégoire, B. Mazure, C. Piette, Extracting muses, in: G. Brewka, S. Coradeschi, A. Perini, P. Traverso (Eds.), ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings, volume 141 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2006, pp. 387–391.

[18] H. van Maaren, S. Wieringa, Finding guaranteed muses fast, in: H. K. Büning, X. Zhao (Eds.), Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings, volume 4996 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 291–304. URL: https://doi.org/10.1007/978-3-540-79719-7_27. doi:10.1007/978-3-540-79719-7_27.

[19] C. Desrosiers, P. Galinier, A. Hertz, S. Paroz, Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems, J. Comb. Optim. 18 (2009) 124–150. URL: https://doi.org/10.1007/s10878-008-9142-4. doi:10.1007/s10878-008-9142-4.

[20] J. P. M. Silva, Minimal unsatisfiability: Models, algorithms and applications (invited paper), in: 40th IEEE International Symposium on Multiple-Valued Logic, ISMVL 2010, Barcelona, Spain, 26-28 May 2010, IEEE Computer Society, 2010, pp. 9–14. URL: https://doi.org/10.1109/ISMVL.2010.11. doi:10.1109/ISMVL.2010.11.

[21] A. Nadel, Boosting minimal unsatisfiable core extraction, in: R. Bloem, N. Sharygina (Eds.), Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23, IEEE, 2010, pp. 221–229. URL: https://ieeexplore.ieee.org/document/5770953/.

[22] V. Ryvchin, O. Strichman, Faster extraction of high-level minimal unsatisfiable cores, in: K. A. Sakallah, L. Simon (Eds.), Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings, volume 6695 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 174–187. URL: https://doi.org/10.1007/978-3-642-21581-0_15. doi:10.1007/978-3-642-21581-0_15.

[23] A. Belov, J. Marques-Silva, Accelerating MUS extraction with recursive model rotation, in: P. Bjesse, A. Slobodová (Eds.), International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011, FMCAD Inc., 2011, pp. 37–40. URL: http://dl.acm.org/citation.cfm?id=2157663.

[24] M. H. Liffiton, K. A. Sakallah, Algorithms for computing minimal unsatisfiable subsets of constraints, J. Autom. Reason. 40 (2008) 1–33. URL: https://doi.org/10.1007/s10817-007-9084-z. doi:10.1007/s10817-007-9084-z.

[25] S. D. Gupta, B. Genc, B. O'Sullivan, Explanation in constraint satisfaction: A survey, in: Z. Zhou (Ed.), Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021, ijcai.org, 2021, pp. 4400–4407. URL: https://doi.org/10.24963/ijcai.2021/601. doi:10.24963/ijcai.2021/601.

[26] M. Gebser, B. Kaufmann, T. Schaub, Conflict-driven answer set solving: From theory to practice, Artif. Intell. 187 (2012) 52–89. URL: https://doi.org/10.1016/j.artint.2012.04.001. doi:10.1016/j.artint.2012.04.001.

[27] A. Felfernig, M. Schubert, C. Zehentner, An efficient diagnosis algorithm for inconsistent constraint sets, Artif. Intell. Eng. Des. Anal. Manuf. 26 (2012) 53–62. URL: https://doi.org/10.1017/S0890060411000011. doi:10.1017/S0890060411000011.

[28] M. Gelfond, V. Lifschitz, Logic programs with classical negation, in: D. Warren, P. Szeredi (Eds.), Proceedings of the Seventh International Conference on Logic Programming (ICLP'90), MIT Press, 1990, pp. 579–597.

[29] P. Simons, I. Niemelä, T. Soininen, Extending and implementing the stable model semantics, Artificial Intelligence 138 (2002) 181–234.

[30] M. Razzaq, R. Kaminski, J. Romero, T. Schaub, J. Bourdon, C. Guziolowski, Computing diverse boolean networks from phosphoproteomic time series data, in: M. Češka, D. Šafránek (Eds.), Computational Methods in Systems Biology, Springer International Publishing, Cham, 2018, pp. 59–74.

[31] R. Kaminski, J. Romero, T. Schaub, P. Wanko, How to build your own asp-based system?!, CoRR abs/2008.06692 (2020). URL: https://arxiv.org/abs/2008.06692. arXiv:2008.06692.

[32] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko, Theory solving made easy with clingo 5, in: M. Carro, A. King, N. Saeedloei, M. D. Vos (Eds.), Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA, volume 52 of *OASIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:15. URL: https://doi.org/10.4230/OASIcs.ICLP.2016.2. doi:10.4230/OASIcs.ICLP.2016.2.

[33] M. Banbara, B. Kaufmann, M. Ostrowski, T. Schaub, Clingcon: The next generation, Theory Pract. Log. Program. 17 (2017) 408–461. URL: https://doi.org/10.1017/S1471068417000138. doi:10.1017/S1471068417000138.