Determining and resolving conflicts in the configuration of high-performance pumps *

Tobias Fischer[1], Christopher Hamkins[2], Sebastian Velten[1], and Pascal Wortel[1]

[1] Fraunhofer Institute for Industrial Mathematics ITWM, Kaiserslautern, Germany
{tobias.fischer,sebastian.velten,pascal.wortel}@itwm.fraunhofer.de
[2] KSB SE & Co. KGaA, Frankenthal, Germany
christopher.paul.hamkins@ksb.com

Abstract. We consider a configuration problem for high-performance pumps, where the configurations that are technically possible are specified in restriction tables. For this problem we propose a *Constraint Programming* based feasibility model and present an algorithm for determining and resolving conflicts. Computational results show that the approach is very well suited for various use cases in practice.

Keywords: Constraint programming, product model, minimal conflicts, conflict resolution

1 Introduction

High-performance pumps and valves are often sold in markets with extremely diverse customer requirements. Although they are assembled from a standardized set of parts kept in stock, hardly any two orders are exactly the same. To be able to fulfill the customer's specification but ensure that the product can be manufactured and that all the technical, pricing, lead time and business limits are respected, a product configuration software is used.

The software is used by sales people to configure and quote the product and to later automatically generate the parts lists and work plans needed in the factory. Sometimes the configuration software will report that there is a conflict with the choices made to satisfy the customer requirements. Due to the complexity of the product model, it is practically impossible for the user to resolve these manually. Automated assistance for conflict resolution is needed.

2 The Configuration Problem

The product is described by a set of features $\mathcal{F}$ with finite domains. The goal is to have a valid configuration with an assignment of a specific value to each

* The project "Development of new methods, algorithms and software for configuration and selection of complex product systems" is supported by a grant from the European Union European Fund for Regional Development.

feature such that all the constraints are fulfilled. The constraints fall into two categories, a consistent product model that stays the same, and a set of user decisions and domain restrictions that vary for each request.

The product structure is a tree-like maximal bill of materials (BOM). The tree nodes can themselves also have sub-nodes, etc., but loops are not allowed and the depth is of course finite. The nodes in the BOM are optional and may or may not exist in a particular configuration depending on feature evaluations.

The main constraints are formulated using restriction tables and conditional assignments. A restriction table simply lists allowed combinations of values for a set of features. A combination of values not corresponding to a line in the table is not allowed. The restriction tables are assigned to nodes and are only active if their node exists. This can lead to nodes being rejected in a solution if the tables they contain cannot be fulfilled due to other constraints. A conditional assignment is an implication that a feature take on a certain value if a condition holds.

A typical model has around 500 features with 4 values on average, 300 relevant nodes, 150 conditional assignments and about 600 restriction tables with 1 to 10 columns each and containing 140,000 lines in total for all tables.

3 The CP Model

We model the configuration problem using a *Constraint Programming* (CP) approach. The CP formulation is the basis for finding feasible configurations (using a backtracking search based on feature and evaluation priorities) and for determining conflicts and resolutions. For the description of the model we use decision variables and constraints provided by Google's OR-Tools CP library (see [2]).

For each feature $f \in \mathcal{F}$ we define an integer decision variable x_f with domain $\{-1, 0, \ldots, |f| - 1\}\}$, where $|f|$ is the number of possible evaluations for f. The value -1 is used to represent intentional nonevaluation of a feature. Furthermore, for each node n in the set of all nodes $\mathcal{N}$ we define a binary decision variable x_n which is equal to 1 if n exists and 0 otherwise.

The most important constraints are the table restrictions. For each node $n \in \mathcal{N}$ let $\mathcal{R}_n$ be the set of its restriction tables. The set of all restriction tables is denoted by $\mathcal{R} = \bigcup_{n \in \mathcal{N}} \mathcal{R}_n$. A restriction table $r \in \mathcal{R}$ constrains the possible combinations of values for p_r features by a set $\mathcal{A}_r$ of q_r vectors of length p_r.

As the nodes are optional and for $n \in \mathcal{N}$ the restriction tables in $\mathcal{R}_n$ need only to be satisfied if n exists, we introduce a *local feature variable* x_f^n for each feature f referenced in a restriction table in $\mathcal{R}_n$. Each of these local variables x_f^n has the same domain as x_f and we add the constraint

$$x_n \leq \left(x_f = x_f^n\right)\texttt{.Var()} \tag{1}$$

to the CP formulation. In this constraint, `(...).Var()` is a binary decision variable which is equal to 1 if the inner constraint is satisfied and 0 otherwise (see [2]). Thus, (1) enforces that x_f and x_f^n are equal if node n exists.

Given the local feature variables, the restriction tables are then represented in the CP formulation by the Google OR Tools constraint

$$\texttt{AllowedAssignment}\left(\left(x^n_{f_1},\ldots,x^n_{f_{p_r}}\right),\mathcal{A}_r\right), \tag{2}$$

which ensures the vector of values assigned to $\left(x^n_{f_1},\ldots,x^n_{f_{p_r}}\right)$ is an element of $\mathcal{A}_r$. In combination with (1) these constraints make sure that the restriction tables are respected if node n exists. Note that the usage of the local feature variables is necessary as `(...).Var()` is not available for `AllowedAssignment(...)` (see [2]).

Apart from the table restrictions, the nodal existence each depend on the values of certain features and other feature values depend on the existence of certain nodes. In the CP model this is reflected by conjunctions and disjunctions of value assignments to feature and node existence variables. Furthermore, some features $f \in \mathcal{F}$ must not take a certain set of values $\mathcal{V}^t_f$ if the feature `OPRTN_PLNTS`, which selects the production plant, is set to a certain value t:

$$\max_{v\in\mathcal{V}^t_f}\left(x_f = v\right)\texttt{.Var()} \leq 1 - \left(x_{\texttt{OPRTN_PLNTS}} = t\right)\texttt{.Var()}. \tag{3}$$

In addition to these constraints, which describe the feasible configurations of the product in general, the CP model can be extended by explicit user requests. The requests either lead to instantiations of decision variables (assignment of a certain value to a feature or determination of node existence) or further domain restrictions for feature variables.

4 Conflict Determination and Resolution

Especially in early stages of product modeling or when users specify many feature assignments, the CP model may be infeasible. In these cases, the determination of the *minimal conflicts* is crucial. They describe the core of the infeasibility and thus form the basis for resolutions that fulfill the specifications as far as possible.

In the following we present a bottom-up approach combined with a binary search for finding all minimal conflicts of a set of constraints. We use this approach since the minimal conflicts are usually rather small. Other procedures for determining all minimal conflicts can for example be found in [3] (top-down approach) and [1] (simultaneous construction of maximal satisfiable and minimal conflicting constraint sets).

Let $\mathcal{C}$ be a set of constraints. $\mathcal{C}$ is a conflict if not all constraints in $\mathcal{C}$ can be fulfilled at the same time. $\mathcal{C}$ is minimal if no proper subset of $\mathcal{C}$ is a conflict. Moreover, let `S` be a set of sets. Then $\mathcal{H}$ is a hitting set of `S` if $\mathcal{S}' \cap \mathcal{H} \neq \emptyset$ for all $\mathcal{S}' \in$ `S`. $\mathcal{H}$ is minimal if no proper subset of $\mathcal{H}$ is a hitting set.

Let `MinC` be the set of minimal conflicts found so far and `H` be the set of minimal hitting sets of `MinC`. Then the bottom-up approach for finding all minimal conflicts of a conflict $\mathcal{C}$ has the following steps:

1. Set $\mathcal{C}' = \mathcal{C}$.

Algorithm 1: Determination of one minimal conflict.

Input: Conflict $\mathcal{C}'$.
Result: Minimal conflict $\mathcal{C}^*$.

```
1  if |C'| = 1 then
2  |  Return C'.
3  Set L := c_1, ..., c_N (ordered constraints of C'), n_min = 0, n_max = N, C* = ∅.
4  while true do
5  |  while true do
6  |  |  Set n* = ⌈(n_max − n_min)/2⌉ and C̃ = C* ∪ {c_1, ..., c_{n*}}.
7  |  |  if C̃ is a conflict then
8  |  |  |  if n* = n_max then
9  |  |  |  |  C* = C* ∪ {c_{n*}} and Break.
10 |  |  |  else
11 |  |  |  |  Set n_min = n*.
12 |  |  else
13 |  |  |  Set n_min = n*.
14 |  if C* is a conflict then
15 |  |  Break.
16 |  else
17 |  |  Set n_min = 0, n_max = n* − 1.
18 Return C*.
```

2. Find a minimal conflict $\mathcal{C}^* \subseteq \mathcal{C}'$ by Algorithm 1. `MinC` := `MinC` $\cup \{\mathcal{C}^*\}$.
3. Update `H` with respect to $\mathcal{C}^*$.
4. For all $\mathcal{H} \in$ `H`:
   - If $\mathcal{C} \backslash \mathcal{H}$ is a conflict: Set $\mathcal{C}' = \mathcal{C} \backslash \mathcal{H}$, go to Step 2.
   - Otherwise: Remove $\mathcal{H}$ from `H`.
5. Return `MinC`.

The update of `H` with respect to $\mathcal{C}^*$ in Step 3 is done as follows:

- Extend each element of `H` by each element of $\mathcal{C}^*$: `H` $:= \{\mathcal{H} \cup c : \mathcal{H} \in$ `H`$, c \in \mathcal{C}^*\}$.
- Remove all duplicate and non minimal hitting sets from `H`.

In addition, note that $\mathcal{C} \backslash \mathcal{H}$ resolves all minimal conflicts of the current set `MinC`. Therefore, if $\mathcal{C} \backslash \mathcal{H}$ is a conflict, this set contains a new minimal conflict and no duplicates are generated. Finally, the approach guarantees that `MinC` is complete (i.e. all minimal conflicts have been found) as in Step 5 the complements $\mathcal{C} \backslash \mathcal{H}$ for all hitting sets $\mathcal{H}$ of `MinC` are feasible.

Given the set of minimal conflicts `MinC` resolutions are determined by relaxing minimal hitting sets of this set and applying the prioritized backtracking search.

5 Use Cases and Computational Results

5.1 Conflict Resolution

In this use case the end user would like to be informed which of the user decisions cause conflicts with the product model and be given suggestions for changing

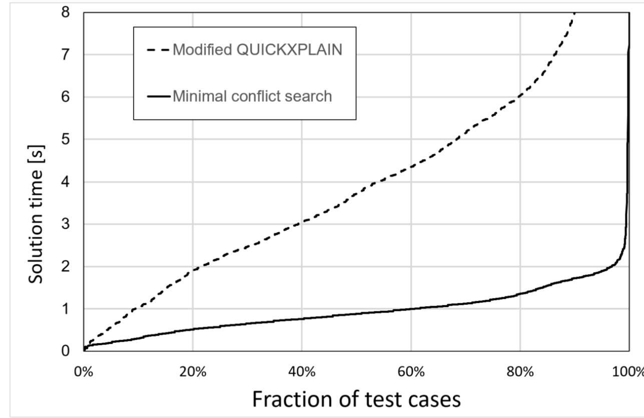

Fig. 1. Solution time for conflict resolution use case

them to get a valid configuration. During product configuration a user proceeds by selecting values for some of the features and the system is to return a valid assignment for the rest of them or to determine that no valid solution exists and return a number of suggestions on how to resolve the problem by changing some of the decisions. Normally the user proceeds feature by feature with the system responding after each choice, so a fast response is very important.

Since this is the same as the use case described in [4] the following variant of this approach was implemented as a baseline. The user decisions were prioritized based on the order in which they were made. Because of the step-by-step decision procedure the QUICKXPLAIN algorithm in [4] simply reports ”relax your last decision” as resolution, rather unsatisfying for the end user. As the product model is soluble without user decision constraints, and each feature value can occur in at least one solution, there are always at least two possible conflict resolutions, so the algorithm was extended to also find all relaxations of single user decisions that resolve the conflict and if there are less than two, also a general relaxation different than ”relax your last decision”.

Fig. 1 shows a comparison of the solution times for the algorithm developed here (Section 4) with the modified QUICKXPLAIN for about 6000 test cases taken from actual and conflicting end user interactions . It can be seen that the new algorithm usually completes in 25% of the time that the baseline required. Further, in general it reports more possible conflict resolutions (potentially one for each minimal hitting set). At the far right of the figure we see that there are a few cases in which both algorithms required substantially more time or didn’t complete, reminding us that the problem at hand remains NP-complete.

5.2 Conflict Analysis

In this use case the product modeler assigns values to some features and finds that no solution is possible, although the feature assignment is intended to be

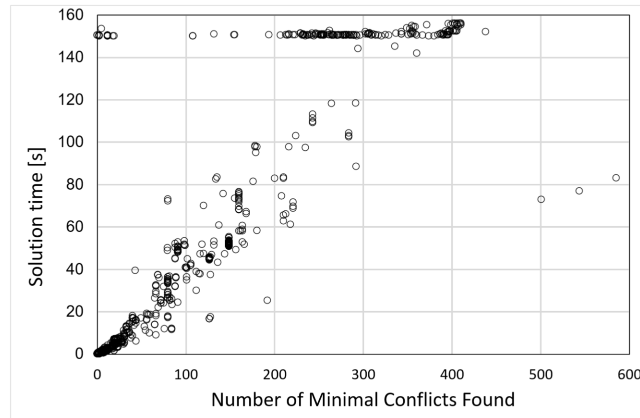

Fig. 2. Running time for conflict analysis use case

a valid configuration. In this case, the product modeler needs to know which of the constraints in the model prevent the assignments to check the formulation of those constraints for errors. The response time is not so important, but the completeness of the answer is. Manual analysis of the conflict usually takes days, so if the system can automatically deliver a complete analysis in less time it is a great benefit for the modeler looking for problems in the model.

The performance of the algorithms for conflict analysis was demonstrated on 3700 test cases with conflicts, also taken from real end user interactions. Fig. 2 shows the processing time required vs. the number of minimal conflicts discovered. The solution time of up to a few minutes is perfectly acceptable for practical use. The large number of minimal conflicts appears surprising at first but can be explained due to (necessary) redundancies in the restriction tables.

For the tests, a time limit of roughly 150 s was enforced. When the time limit was reached, the algorithm had always reported some minimal conflicts. Since every single minimal conflict needs to be addressed to resolve the overall conflict, reporting even one minimal conflict is a help for the product modelers.

References

1. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization, in: Manuel V. Hermenegildo, Daniel Cabeza (Eds.), Practical Aspects of Declarative Languages, Springer 2005, 174–186.
2. Google. Google's OR-Tools, 2020, https://developers.google.com/optimization/
3. Han, B., Lee, S.-J.: Deriving minimal conflict sets by CS-trees with mark set in diagnosis from first principles, IEEE Trans. Syst. Man Cybern. B 29 (2) 1999, 281–286.
4. Junker, U.: QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. Proceedings of the 19th National Conference on Artificial Intelligence, 16th Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA, pp. 167–172.