Proceedings of the

ECAI 2006
Workshop on Configuration

affiliated with the

17th European Conference on
Artificial Intelligence

August 28-29, 2006
Riva del Garda, Italy

Carsten Sinz
Albert Haag
(Eds.)

Dedicated in memory of
Daniel Mailharro

Preface

Product configuration has a long-standing tradition in AI research. Starting in the 80s with work on configuration of large computer systems, it has developed into a flourishing area for both researchers and practitioners. Powerful knowledge-representation models are necessary to capture the great variety and complexity of configurable products. Efficient reasoning methods are required to provide intelligent interactive behavior in configuration software, such as solution search, satisfaction of user preferences, optimization, diagnosis, or explanation.

Today, a large number of commercial configuration solutions is available on the market, covering conventional configuration of materials and equipment as well as software and service configuration (like financial portfolio configuration, insurance optimization, or travel advisors). Configuration is more than ever a challenging area for applying novel AI techniques since more and more sophisticated reasoning tasks are delegated to the configurator software; the software thus has to integrate product-assembly knowledge along with customer classification, adaptive sales strategies, and customer assistance. This integration becomes particularly critical for e-business applications where customers directly configure products through the Internet with no professional assistance and without a deep knowledge of the products they intend to buy.

This workshop continues the series of eight successful Configuration Workshops started at the AAAI'96 Fall Symposium and continued on IJCAI, AAAI, and ECAI conferences since 1999. Moreover, special issues on configuration in the AI-EDAM '98, IEEE Expert '98, and AI-EDAM 2003 journals demonstrate the continuous successful work in this field. Besides the participation of researchers from a variety of different fields, past events always attracted significant industrial interest from major configurator vendors like ILOG, Oracle, or SAP, as well as from industries with applications like ABB, DaimlerChrysler, HP, or Siemens. The goal of these workshops is to promote high-quality research in all technical areas related to configuration and to bring together researchers and practitioners from industry and academia.

These working notes present contributions dealing with various topics closely related to configuration problem modeling and solving. Eleven papers and four extended abstracts demonstrate both the wide range of applicable AI techniques and the diversity of the problems and issues that need to be studied and solved to construct and implement effective configuration solutions.

Carsten Sinz
Albert Haag

August 2006

Organization

Chairs

Carsten Sinz, Johannes Kepler Universität Linz, Austria
Albert Haag, SAP AG, Germany

Organizing Committee

Claire Bagley, Oracle Corporation, USA
Alexander Felfernig, Universität Klagenfurt, Austria
Esther Gelle, ABB Corporate Research AG, Switzerland
Barry O'Sullivan, University College Cork, Ireland

Program Committee

Michel Aldanondo, Ecole des Mines d'Albi, France
Claire Bagley, Oracle Corporation, USA
Boi Faltings, EPFL, Switzerland
Alexander Felfernig, Universität Klagenfurt, Austria
Felix Frayman, Wizdom Technologies LLC, USA
Gerhard Friedrich, Universität Klagenfurt, Austria
Esther Gelle, ABB Corporate Research AG, Switzerland
Laurent Henocque, Université de la Méditarranée, France
Dietmar Jannach, Universität Klagenfurt, Austria
Ulrich Junker, ILOG S.A., France
Michael Koch, TU München, Germany
Diego Magro, Universita di Torino, Italy
Tomi Männistö, Helsinki University of Technology, Finland
Sanjay Mittal, Selectica Inc., USA
Klas Orsvarn, Tecton System AB, Sweden
Barry O'Sullivan, University College Cork, Ireland
Frank Piller, MIT, USA
Marty Plotkin, Oracle Corporation, USA
Mihaela Sabin, Rivier College, USA
Markus Stumptner, Advanced Computing Research Center, Australia
Markus Zanker, Universität Klagenfurt, Austria

Additional Reviewers

Tomas Axling, Olivier Lhomme

Table of Contents

Full Papers

Short Papers and Position Statements

Extended Abstracts

Full Papers

Constraint and Variable Ordering Heuristics for Compiling Configuration Problems

Nina Narodytska[1] and **Toby Walsh**[2]

Abstract. We develop two heuristics for reducing the time and space required to obtain a binary decision diagram representation of the solutions of a configuration problem. First, we show that dynamics of the growth in the size of the decision diagram depend strongly on the order in which constraints are added and propose a heuristic constraint ordering algorithm that uses the distinctive clustered hierarchical structure of constraints graphs of configuration problems. Second, we propose a variable order heuristic based on the typical clustered structure of the constraint graph to be used by a variable sifting algorithm during the BDD construction procedure.

1 INTRODUCTION

In configuration problem, we wish to determine a customised product configuration based on a set of pre-defined components and a set of constraints and requirements describing possible component compositions [8]. Product configuration is often an interactive procedure, where a customer sets desired product characteristics and receives feedback from the configurator about valid values of remaining characteristics. The interaction continues until a complete and valid configuration is found. Such a scenario requires an efficient mechanism to ensure the current decisions can be consistently extended.

One way to achieve this is to use a two-phase approach, where in the first, offline phase the product constraints are compiled into a compact representation that completely describes the set of all possible product configurations. This representation can be efficiently manipulated by the interactive configurator during the second stage. Hadzic et.al. [6] suggested using Binary Decision Diagrams (BDDs) [2] to represent the space of valid product configurations and showed that this approach enabled efficient interactive configuration process.

In this paper, we focus on optimising the first phase of the configuration process — compilation of the product rules into a BDD. This is computationally hard task that can require significant CPU time and memory resources. Although it is performed offline and, therefore, is not required to provide interactive real-time responses, performance is still important, because for large configuration problems BDD construction can demand more space and time resources than are available.

We propose two techniques for improving speed and memory consumption of the BDD construction procedure for configuration problems. First, we suggest a heuristic for constraint ordering that uses the distinctive clustered hierarchical structure of configuration problems constraint graphs. Second, we propose a variable ordering heuristic to apply during the variable sifting procedure [10] which groups problem variables based on the clustered nature of the constraint graph. The combined use of these techniques results in a one to two orders of magnitude reduction in the time to construct BDDs for problems from the configuration benchmarks suite [3].

[1] National ICT Australia, Australia, email: n.naroditskaya@gmail.com

[2] National ICT Australia and School of Computer Science and Engineering, University of New South Wales, Australia, email: Toby.Walsh@nicta.com.au

2 THE STRUCTURE OF CONFIGURATION PROBLEMS

A straightforward approach to translating the configuration problem description into BDD is to represent every rule as a separate BDD and conjoin together these BDDs. If problem variables have multi-valued domains, BDDs can be replaced by Multi-Valued Decision Diagrams (MDD). When we applied this approach to a number of configuration benchmarks [3], we noticed that the size of BDDs grew in a quite unusual pattern. For example, the graph in Figure 1 shows the relationship between the number of constraints added and the number of nodes in the resulting BDD for the Renault Megane configuration benchmark [3]. Interestingly, in this example the BDD grew almost monotonically in size, except at the end, where the addition of critical constraints caused a dramatic drop in size. Other benchmarks demonstrated similar behaviour.

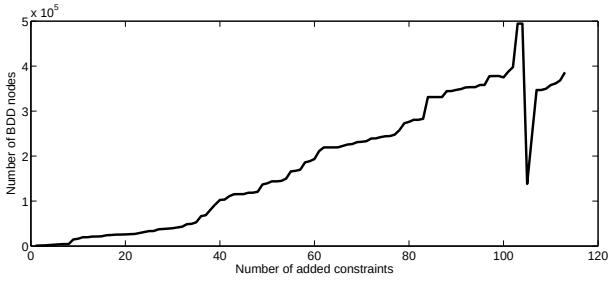


Figure 1. Dynamics of BDD growth for the Renault Megane configuration benchmark. Constraints were added in the order specified in the benchmark.

This contrasts with what we typically observe when applying BDD to solving combinatorial AI problems. For example, Figure 2 shows a similar graph for the Langford's numbers problem [4]. In this case, the size of the BDD is an order of magnitude bigger at an intermediate point than at the end. This means that BDDs obtained in intermediate steps may require more memory than is available even though the final BDD representing all problem solutions may be comparatively small. Large BDD size has negative impact not only on the

memory consumption but also on the time of construction of the final BDD containing all problem solutions, since the complexity of adding every following constraint is equal to $O(|B_{i-1}| \cdot |b_i|)$, where $|B_i|$ and $|b_i|$ are the number of nodes in the aggregate BDD after adding first $i-1$ constraints and the number of nodes in the BDD representing the ith constraint, respectively.

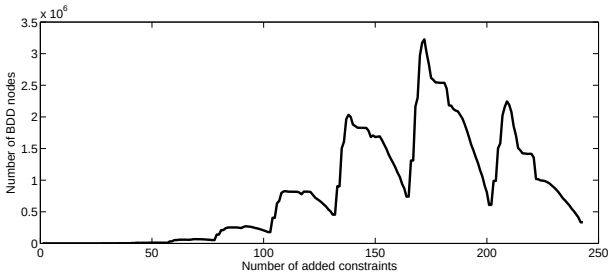


Figure 2. Dynamics of BDD growth for the Langford(11,2) problem

The monotonic growth in BDD size observed for configuration problems is therefore a highly desirable property both from the memory consumption and speed points of view. We therefore tried to determine the properties of configuration problems which are responsible for this monotonic behaviour. Our goal is to use these properties in a more directed manner in order to improve further the performance of the BDD construction process.

In order to investigate further the properties of configuration problems, we use their weighted primal constraint graphs. A primal constraint graph of a problem is an undirected graph where nodes correspond to problem variables and edges describe constraints. Two variables are connected by an edge iff they participate in at least one common constraint. The weight of the edge is equal to the number of common constraints involving these two variables. Figure 3 shows a typical configuration benchmark constraint graph.[3] We notice that this graph has a distinct tree-like skeleton: it is formed by a tree of clusters, most of which have a star-like structure, with a few central nodes connected to a large number of peripheral nodes. In contrast, the constraint graph of the Langford's numbers problem constitutes a clique.

The tree-like structure of constraint graphs can help explain the monotonic behaviour of BDD. Consider an idealised problem whose constraint graph is a tree. All problem variables are binary. Problem constraints are also binary and selected in such a way that the constraint graph is arc consistent (that is, we avoid those binary constraints that assign a fixed value to one of their variables). It is easy to show that for such an idealised problem it is possible to construct an order in which to add constraints to the aggregate BDD which will guarantee monotonic growth in the size of the BDD. Any constraint ordering that adds exactly one new variable to the aggregate BDD on every step (starting from step 2) has this property. Figure 4a shows the result of applying such an ordering to a randomly generated tree-structured problem. It should be noticed that not every possible constraint ordering leads to monotonic BDD growth. For example, for the same problem, adding constraints in a random order

[3] Most configuration benchmark definitions contain a large number of unary constraints. In order to obtain more meaningful results, we got rid of these constraints by enforcing arc consistency for all problems in our experiments. The constraint graph on the figure shows the structure of the problem after enforcing arc consistency.

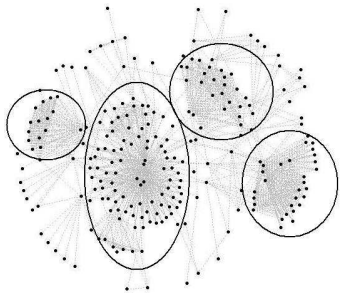


Figure 3. Constraint graph of a Mercedes configuration benchmark (C211_FS). The circles indicate major clusters identified by Markov CLuster Algorithm(MCL)

gave the growth shown in Figure 4b.

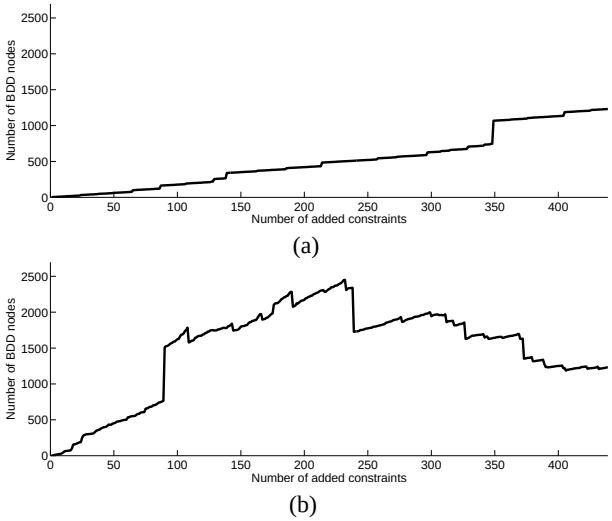


Figure 4. Dynamics of BDD growth for a randomly generated tree-structured problem. (a) shows monotonic BDD growth obtained using a special constraint ordering; (b) shows non-monotonic growth yielded by random constraint ordering

3 CONSTRAINT ORDERING HEURISTIC

Based on these observations, we proposed a heuristic for adding constraints that attempts to ensure monotonic growth in the size of the BDD, and to keep the size of the aggregate BDD as small as possible on every step. This heuristic is based on the following guidelines:

- **To respect the tree-like structure of many configuration problems**. In particular, we want the heuristic to guarantee monotonic growth in the size of the BDD in the extreme case when the constraint graph is a tree.
- **To respect the clustered structure of many configuration problems**. In addition to the tree-like structure, many configuration problems also have a strong clustered structure.
  For example, the circles on Figure 3 indicate major clusters identified by Markov CLuster Algorithm(MCL) [12] for the C211 FS benchmark. A cluster typically corresponds to a group of variables which are logically very tightly connected. For example,

they might describe a single component of the product, e.g., the engine of the car. We observed that algorithms that add most or all of the constraints connecting variables of a cluster before moving to other clusters are faster than algorithms that do not respect the clustered structure of the constraint graph. Adding all the constraints connecting a group of variables may reduce the number of valid combinations of these variables and thus the size of the BDD.

- **To keep the number of variables small**. Typically, BDD size grows with the number of variables added. Therefore, before adding new variables to the BDD, we should try to add all the constraints that involve variables already contained in the BDD. If this is not possible, we should select a constraint that adds as few new variables as possible.

Among the many heuristic algorithms that we implemented and evaluated, the following algorithm, further referred to as Algorithm 1, produced the best results for the majority of the benchmarks. Figure 5 shows the flowchart of the algorithm. The internal state of the algorithm consists of a list of constraints already added to the BDD, a list of remaining constraints, a list of variables already added to the BDD, a list of remaining variables and a stack of variables that have been used as central variables (the current central variable is located at the top of the stack).

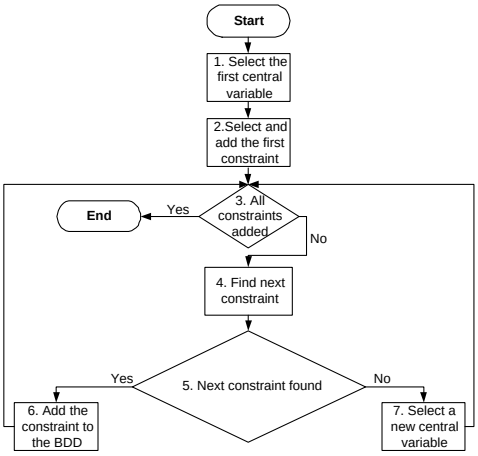


Figure 5. Flowchart of Algorithm 1

Step 1. Selection of the first central variable. Among all problem variables, a variable whose adjacent edges have the largest total weight is selected to be the first central variable. This variable is stored at the top of the stack of central variables.

Step 2. Selection of the first constraint. Among all constraints that include the central variable, a constraint with the biggest number of variables in its scope is selected.

Step 4. Selection of the next constraint. On this step, the next constraint to add to the BDD is selected from the set of remaining constraints.

4.1 All constraints that contain the current central variable are selected among the remaining constraints. If no such constraints exist, then Step 4 terminates without selecting a candidate constraint.

4.2 From the obtained set of constraints, all constraints that contain the smallest number of variables not yet added to the BDD are selected.

4.3 From the obtained set of constraints, constraints containing the smallest number of variables are selected.

4.4 For each selected constraint, the sum of weights of adjacent edges of all its variables is computed and constraints with the largest sum are selected. The first such constraint becomes the next candidate for being added to the BDD.

Step 7. Selection of the next central variable.

7.1 The set of all neighbours of the current central variable is computed (the current central variable is the variable located at the top of the stack of central variables).

7.2 From the obtained set of variables, all variables that do not participate in scopes of any of the remaining constraints are eliminated.

7.3 If the obtained set of variables is empty, the current central variable is popped from the stack of central variables and the algorithm returns to step 7.1.

7.4 From the obtained set of variables, a variable whose adjacent edges have the largest total weight is selected to be the next central variable. This variable is stored at the top of the stack of central variables. If there are several such variables, the first in the original variable ordering is selected.

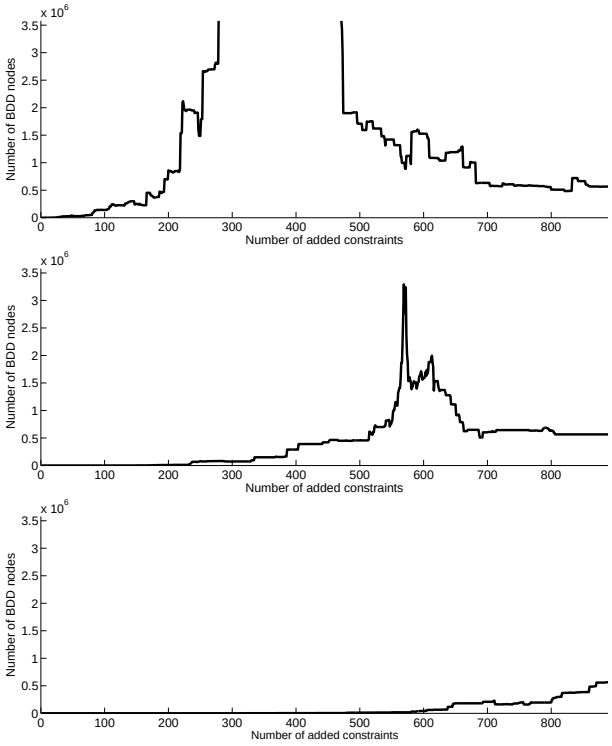
This algorithm selects a central variable and adds all constraints involving this variable, trying to add as few new variables as possible on every step. Among all constraints that passed filters 4.1 to 4.3, a constraint containing the most influential variables is selected, since such constraint is more likely to reduce BDD size. Then the next central variable is selected among the neighbours of the current central variable. We select the heaviest variable hoping that it will be the centre of the next cluster (which is usually the case in configuration problems).

Figure 6 shows the results of applying this algorithm to two problems from the configuration benchmarks. The graphs compare the dynamics of the BDD growth when adding constraints in a random order, in the original order specified in the benchmark description, and in the order produced by Algorithm 1. In both cases, the order produced by Algorithm 1 resulted in almost monotonic growth of the BDD and kept the BDD size smaller than the other two orderings on all steps. As a result, this algorithm significantly reduced the time of BDD construction. We obtained similar results for all other problems from the configuration benchmarks suite, which we were able to solve without the dynamic variable reordering optimisation introduced in the following section.

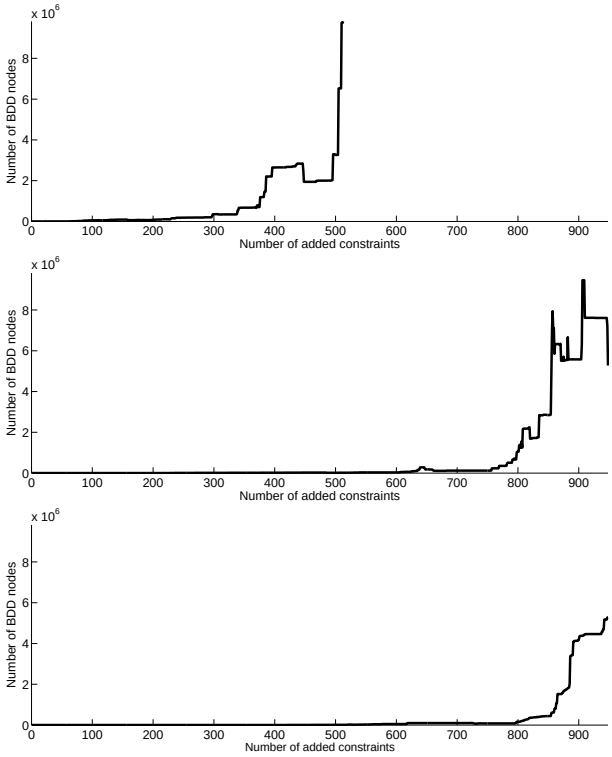
Another interesting feature of the graphs is that the original constraint ordering was much more efficient than the random ordering. This can be explained by the fact that in most cases the original ordering reflects the natural structure of the problem. For example, it typically groups together constraints describing a single component of the product. Our constraint ordering heuristic was able to make further improvements to this order.

4 VARIABLE ORDERING HEURISTIC

In the previous section, we showed that in configuration problems it is often possible to achieve monotonic growth in the size of the BDD using constraint ordering heuristics. However, for large configuration



(a)



(b)

Figure 6. Dynamics of BDD growth when adding constraints in a random order, the original order, and in the ordering produced by Algorithm 1 for (a) the C211_FS configuration benchmark and (b) the C638_FVK configuration benchmark

problems even a monotonically growing BDD can still be very large. Therefore, in order to reduce the space and time requirements of these problems, we would like to find ways to reduce further BDD size.

One common technique to achieve this is based on dynamic variable reordering. A BDD constitutes a multi-level directed graph, where every level corresponds to a single variable. It is well-known that BDD size depends significantly on the assignment of variables to levels, or, in other words, on the order of variables [2]. By reordering variables, it is often possible to reduce dramatically the size of a BDD. Variable reordering can be applied either to the final BDD, in order to reduce the representation of the solution space of the problem, or during BDD construction, after adding only part of the constraints, in order to reduce the intermediate BDD size and speed up addition of the remaining constraints.

Determining the optimal variable ordering is a co-NP-complete problem [1]; therefore in practice approximations to the optimal ordering are used. Rudell [10] has proposed a sifting algorithm for dynamic variable reordering and demonstrated that it allowed achieving significant reduction of BDD size for some types of constraint satisfaction problems. The idea of the sifting algorithm is to move each variable up and down in the order to find a position that provides local minimum of the BDD size. This procedure is applied to every problem variable sequentially. We applied the sifting algorithm provided by the CUDD package [5] to configuration benchmarks. We used adaptive threshold to trigger the variable reordering procedure. Whenever the BDD size reached the threshold, we performed variable reordering and, if the reduced BDD was bigger than 60% of the current threshold, the threshold was increased by 50%. The initial threshold was equal to 100000 BDD nodes.

We combined variable sifting with different constraint orderings described in Section 3. Table 1 below summarises the results of these experiments. As can be seen from the table, dynamic variable reordering significantly reduces the time of BDD construction for all constraint orderings.

The efficiency of dynamic variable reordering can be further improved by using knowledge about the problem structure. First, we observe that locating strongly dependent variables close to each other typically reduces the BDD size.

For problems with strongly pronounced clustered structure, variables comprising a cluster should therefore be kept close to each other in the variable ordering. Second, Panda and Somenzi [9] noticed that dependent variables tend to attract each other during variable sifting, which results in groups of dependent variables being placed in suboptimal positions. In order to avoid this effect, dependent variables should be kept in contiguous groups that are moved as a single variable during variable sifting.

We suggest using the clustered structure of configuration problems to identify groups of dependent variables. We modified the previous algorithm to partition problem variables into contiguous groups corresponding to clusters identified by MCL before starting the BDD construction process. However, it is important to put only strongly connected variables in the same group and avoid grouping weakly connected variables. Therefore, among the clusters found by the MCL algorithm, we only group clusters that have projections bigger than 0.35. The projection of a cluster is determined as the average over all variables in the cluster of the ratio of the total amount of edge weights for the variable within the cluster to the overall amount of edge weights for the variable [12]. Grouped variables are kept contiguous by the reordering procedure. In addition, we allow variables within the group to be reordered before performing the group sifting.

Note that performance of the variable sifting algorithm significantly depends on the initial order of variables. In all cited experiments, we used the initial order specified in the benchmark description. In particular, when performing variable grouping, variable ordering within a group was selected based on the benchmark order and position of the group in the total variable order was selected based on the position of its first variable in the benchmark. Interestingly, replacing this variable order with random order often dramatically increased the time of BDD construction.

Figure 7 shows the behavior of the new variable reordering algorithm in combination with different constraint orderings. Table 1 compares the performance of different combinations of constraint and variable reordering algorithms proposed in this paper.

5 EXPERIMENTAL RESULTS

This section compares the performance of the proposed techniques for problems from the configuration benchmarks suite. The algorithms were implemented in C++ using the CUDD 2.3.2 BDD package from the GLU 2.0 library [5] and the implementation of the MCL algorithm obtained from [7]. In all experiments, the MCL algorithm was used with the inflation parameter set to 5. The inflation parameter affects cluster granularity; we select it to be equal to 5 to get fine-grained clusterings. Experiments were run on a 3.2GHz Pentium 4 machine with 1GB of memory. Table 1 represents results of our experiments.

In most cases, MCL produced satisfactory cluster decomposition. However, it failed in several large problems, namely, C209 FA, C210 FVF, C211 FW, C638 FKA. In these problems, the majority of variables are grouped into clusters, but there are also several variables connected to virtually every problem variable. MCL identifies such constraint graph as a single large cluster. Therefore, whenever MCL encounters a cluster that contains more than half of problem variables, we remove its central variables, that are the heaviest variables in the cluster, and repeat clusterisation.

As can be seen from the table, constraint ordering constructed according to Algorithm 1 (column 10) reduces time of BDD construction compared to random (column 4) and, in most cases, the original (column 7) constraint ordering. It should be noticed that original constraint ordering typically produces quite good results too. This is due to the fact that the original ordering typically follows the problem structure very closely. For example, all constraints describing a single component of the product are typically placed contiguously in the original ordering. On the other hand, Algorithm 1 is able to find a good constraint ordering even if the constraints are randomly shuffled. Comparing columns 5 and 6, 8 and 9, 11 and 12, we can see that variable ordering heuristic based on grouping clustered variables reduces BDD construction time compared to pure variable sifting for the majority of benchmarks.

6 RELATED WORK

The idea of using BDDs to represent configuration problems solution space has been proposed by Hadzic et. al. [6]. They were mainly concerned with reducing the size of the final BDD in order to improve real-time responsiveness of the configurator and did not care much about efficiency of BDD construction. In contrast we focus on reducing time and memory requirements of BDD construction procedure. Therefore, our first improvement is targeted towards optimising the ordering of adding constraints to the BDD, which does not effect the

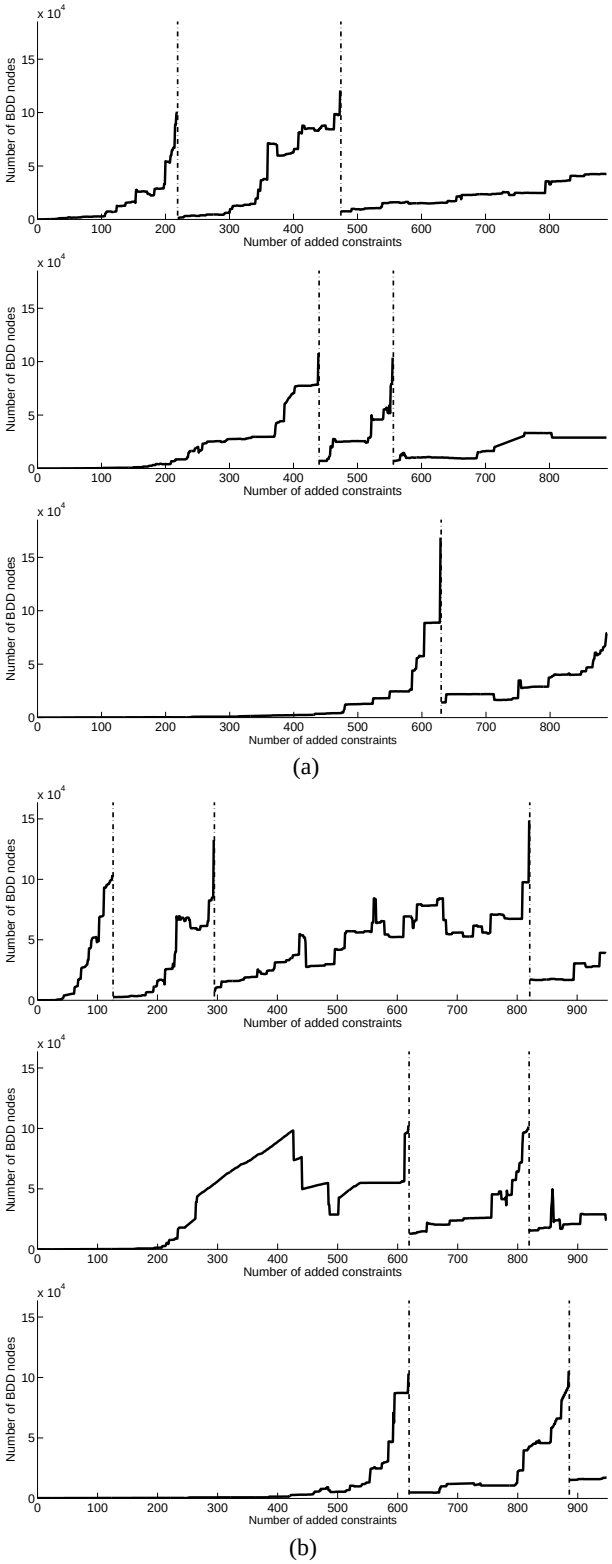


Figure 7. Dynamics of BDD growth when applying dynamic variable reordering and variable grouping in combination with different constraint orderings (random ordering, the original ordering, ordering produced by Algorithm 1) for (a) the C211_FS configuration benchmark, (b) the C638_FVK configuration benchmark. Dashed lines indicate instances when the BDD size reached threshhold and dynamic variable reordering took place.

Table 1. Comparison of different BDD construction algorithms. Columns 4 to 12 show CPU time spent on BDD construction in seconds. The run-time of MCL algorithm is included. "–" denotes that the given problem could not be solved by the corresponding algorithm either because the BDD size exceeded 15,000,000 nodes or because it was interrupted after 10,000 seconds.

Benchmark	# of variables[4]	# of constraints	Random constraint ordering			Original constraint ordering			Ordering produced by Algorithm 1		
			no var reordering	variable sifting	variable sifting + grouping	no var reordering	variable sifting	variable sifting + grouping	no var reordering	variable sifting	variable sifting + grouping
Renault	99	112	54	37	37	30	36	36	**25**	27	27
C169 FV	39	76	0.02	0.02	0.02	0.02	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**
D1119 M23	47	178	0.14	0.14	0.1	0.07	0.07	**0.05**	0.07	0.07	**0.05**
C250 FW	123	321	0.3	0.3	0.12	0.15	0.15	0.09	0.08	0.08	**0.07**
C211 FS	238	889	1432	70	18	231	58	17	48	19	**14**
C638 FVK	426	948	–	69	36	407	59	47	259	52	**23**
C638 FKB	495	1512	–	–	731	–	9442	150	–	522	**135**
C171 FR	441	1775	–	–	–	–	6590	**794**	–	–	1602
C210 FVF	489	1849	–	–	–	–	4612	3779	–	–	**1261**
C209 FA	492	1939	–	–	–	–	**634**	–	–	3858	1097
C211 FW	337	3188	–	–	–	–	–	–	–	**1380**	3813
C638 FKA	517	5272	–	–	4523	–	–	2796	–	3761	**390**

size of the final BDD, but reduces intermediate BDD size and memory consumption.

Sinz [11] proposed an alternative approach to configuration problems precompilation based on construction of optimal set of prime implicates of the problem rules.

The variable sifting algorithm for dynamic variable reordering was proposed in [10]. Panda and Somenzi [9] noticed that dependent variables tend to attract each other during variable sifting and suggested that this effect could be eliminated by grouping dependent variables and moving them as a single variable during variable reordering. They developed the group sifting algorithm, which automatically finds and groups dependent variables. However, this algorithm does not take into account the structure of the problem. Therefore, in our experiments (not cited here), it did not perform as well as the variable grouping algorithm described in Section 4.

7 CONCLUSIONS AND FUTURE WORK

This paper has proposed two heuristics for reducing the time and space required to obtain a BDD representation of the solutions to a configuration problem. We first showed that the dynamics of BDD growth depends strongly on the order in which constraints are added to the BDD and proposed a constraint ordering heuristic based on the tree and clustered structure of the primal constraint graph of many configuration problems. We then showed that configuration problems can benefit from dynamic variable reordering and proposed a way to further improve the efficiency of dynamic variable reordering by grouping variables based on the decomposition of the primal constraint graph of the problem into clusters.

As a part of the ongoing work, we are developing static variable ordering heuristics to further improve performance of BDD compilation. In addition, we are evaluating applicability proposed heuristics in other application domains, in particular, in verification problems. Finally, it would be interesting to establish relation between problem structure and performance of proposed heuristics. For example, it is important to understand why variable grouping decreases performance of BDD construction for some problems and to be able to predict such behaviour.

REFERENCES

[1] Beate Bollig and Ingo Wegener, 'Improving the variable ordering of OBDDs is NP-complete', *IEEE Transactions on Computers*, **45**(9), 993–1002, (1996).

[2] Randal E. Bryant, 'Graph-based algorithms for Boolean function manipulation', *IEEE Transactions on Computers*, **35**(8), 677–691, (1986).

[3] CLib: configuration benchmarks library. `http://www.itu.dk/research/cla/externals/clib`.

[4] Langford's number problem specification in the CSPLib library. `http://csplib.org/prob/prob024/index.html`.

[5] GLU BDD packages. `ftp://vlsi.colorado.edu/pub/vis/`.

[6] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune Møller Jensen, Henrik Reif Andersen, Henrik Hulgaard, and Jesper Møller, 'Fast backtrack-free product configuration using a precompiled solution space representation', in *Proceedings of the International Conference on Economic, Technical and Organizational aspects of Product Configuration Systems*, pp. 131–138, (2004).

[7] Implementation of the MCL algorithm. `http://micans.org/mcl/`.

[8] Sanjay Mittal and Felix Frayman, 'Towards a generic model of configuration tasks', in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 1395–1401, (1989).

[9] Shipra Panda and Fabio Somenzi, 'Who are the variables in your neighborhood', in *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pp. 74–77, (1995).

[10] Richard Rudell, 'Dynamic variable ordering for ordered binary decision diagrams', in *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design*, pp. 42–47, (1993).

[11] Carsten Sinz, 'Knowledge compilation for product configuration', in *Configuration Workshop Proceedings, 15th European Conference on Artificial Intelligence (ECAI-2002)*, pp. 23–26, (2002).

[12] Stijn van Dongen, 'A cluster algorithm for graphs', Technical Report INS-R001, National Research Institute for Mathematics and Computer Science, Netherlands, Amsterdam, (2000).

[4] after enforcing arc consistency

A Survey of Explanation Techniques for Configurators

Albert Haag[1] and **Ulrich Junker**[2] and **Barry O'Sullivan**[3]

Abstract. Explanations are an essential feature of interactive configurators. They allow a user to identify the reasons for a failure and to understand the presence or absence of features in intermediate configurations. In spite of numerous publications about explanation techniques, a survey on explanation methods for configuration has, so far, been absent from the literature. In this paper, we give an abstract definition of the basic explanation capabilities of a configurator and review two of the principal explanation techniques used in industrial configurators, namely Truth Maintenance Systems (TMS) and QuickXplain. We conclude the survey with a list of advanced issues and topics for future research.

1 Introduction

Explanations have always played a crucial role in configuration. In their definition of the configuration task, Mittal and Frayman [23] state that an explanation of failure should be computed if the given user requirements cannot be fulfilled. Furthermore, explanations allow a user to understand why features have been selected or eliminated during the configuration process.

We illustrate the importance of explanations for a simple car configuration problem involving features such as the type (e.g. berline, convertible, station-wagon), the model (e.g. basic, standard, luxury), the engine type (petrol or diesel), the colour, and other diverse options (e.g. ABS, air-conditioning, roof-rack, sunroof, CD, TV, automatic gearbox). Some features are incompatible with each other. For example, it is not possible to choose a roof-rack when a sunroof or a convertible has been selected. Furthermore, some features necessitate others. For example, suppose that a luxury model pre-selects the options CD, airbag, ABS, and air-conditioning, and that a standard model requires ABS. A configurator needs to satisfy these compatibility and requirement constraints, given user choices, by adding or removing features. Suppose a "layman" user has selected a luxury model, berline, sunroof, TV, and ABS. As the user has no detailed knowledge of the compatibilities and requirements, he might ask why a feature such as air-conditioning has been selected in the configuration. An explanation for the presence of a feature comprises the user choices that require the inclusion of the feature. In the example, air-conditioning has been included since the user has chosen a luxury model. Furthermore, the user may ask why a feature such as a roof-rack has been eliminated in the first configuration step. An explanation for the absence of a feature comprises the user choices that are incompatible with the feature. For example, a roof-rack has been eliminated because the user selected a sunroof.

Finally, it is possible that the user's choices cannot be fulfilled by the given catalog. For example, suppose that the user wants a CD, TV, GPS, and automatic gearbox, which consume 5, 10, 10, 30 Watts, respectively. If total power consumption must be less than 40 Watts, then these user choices cannot be fulfilled. An explanation of failure is a minimal unsatisfiable subset of the user choices. For example, the TV and automatic gearbox form one explanation of failure, while GPS and automatic gearbox form another.

There have been many technical papers about explanation [1, 12, 16, 24, 30] and most industrial configurators incorporate an explanation facility. Nevertheless, a comprehensive definition of the explanation problem and a survey of the principal approaches has been missing. This survey paper seeks to give the relevant definitions and to describe the major techniques for finding explanations. Explanations, as considered in this paper, are completely declarative and are independent of a particular configuration technique and of a particular computation sequence. We use a declarative formulation of the underlying configuration problem as is typical in approaches based on description logics, SAT, constraint programming, and integer programming. We assume that the configuration problem is specified by a set of constraints from a suitable logical language [6, 17]. A configuration is a conjunction of facts (such as value assignments to attributes) that is consistent with respect to the constraints. It is important that this logical characterisation precisely describes the behaviour of the given configurator. In particular, the logical characterisation may lack axioms about the completeness of configurations since the configurator may return an incomplete configuration without checking whether it can be completed.

There are two main paradigms for computing explanations in configuration. The first one involves recording proofs computed by a problem solver whilst the second approach involves performing multiple consistency checks. Both methods are independent of the form of the constraints. They can be applied in the context of SAT, constraint programming, description logics, etc. We exemplify these methods using truth-maintenance systems (TMS) [4, 5] and QuickXplain [16], since these methods are not only of algorithmic interest, but have proved their effectiveness in industrial configuration tools and served as examples for others.

The survey is organised as follows. We first define the explanation capabilities of configurators based on an analysis of different user roles (Section 2). Then we present two major techniques for finding explanations, namely TMS (Section 3) and QuickXplain (Section 4). Finally, we discuss several advanced issues and identify topics for future research (Section 5).

2 Basic Explanation Capabilities

Given a set of user choices, a configurator may produce a configuration that can be incomplete, or report an inconsistency. In both cases, the user may need assistance on how to proceed. This assistance can be provided by an explanation capability that is able to answer typical

[1] SAP AG, Germany (albert.haag@sap.com)
[2] ILOG S.A., France (ujunker@ilog.fr)
[3] 4C, University College Cork, Ireland (b.osullivan@cs.ucc.ie)

questions based on the underlying product model and the reasoning process that was performed by the system. The precise form of the explanation depends on the *class of user* at hand. In configuration, we can distinguish between modelers, expert users, and end-users, all having different expectations regarding explanation.

Modelers can use explanations as one convenient way to test their configuration model. This model might be inconsistent, meaning that the constraints associated with a component type cannot be satisfied. These inconsistencies can be caused by modelling errors, which are difficult to isolate if constraint sets are large and the constraints are complex. Explanations are a good way to isolate problematic subsets of constraints. In this case, the explanation should contain the constraints of the configuration model.

An *expert user* might be interested in detailed information about the features that have been selected or eliminated by a configurator in a B2B scenario. The explanation should consist of user requirements, defaults, and possibly constraints of the configuration model.

An *end-user* will require guidance on how to select desired features, to avoid undesired ones, and to resolve inconsistencies. This situation arises in B2C scenarios, as addressed by public web-based configurators. The user enters requirements sequentially. Each new requirement reduces the possible choices for the remaining features. As the end-users do not have detailed product knowledge, they can easily state inconsistent requirements. In this case, an explanation of failure needs to highlight the problematic user choices.

The user's role, thus, determines which constraints appear in explanations and which should be hidden. The general rule is that an explanation should only contain elements that the user can modify.

We now give a general definition of an explanation problem. We characterise an explanation problem by a *background*, a *foreground*, and a *query*. The foreground contains the constraints that can appear in an explanation such as the user requirements (and defaults) in a B2B or B2C configuration scenario or the constraints of the configuration model in a debugging scenario. The background contains all constraints that are needed to find an explanation, but must not appear therein. The background, thus, depends on the scenario (B2B, B2C, or debugging). For example, the background for a B2C scenario might contain all the constraints of the configuration model. The query depends on the question posed:

- What are the reasons for inconsistency?
- Why is a particular feature selected by the system?
- Why is a particular feature unavailable for selection by the user?

In the first case, the query is the inconsistency. In the second case, it is the presence of a feature. In the third case, it is the absence of the feature. The second and third problems can be mapped to the first problem by adding the negation of the query to the background.

An *explanation* is a minimal subset of the foreground that logically entails the query. Minimality means that the query is no longer entailed if some elements are removed from the explanation. This is particularly important if the configurator should give guidance for restoring consistency, for avoiding an undesired feature, or for recovering an excluded feature. In all these cases, the configurator will determine an adequate explanation and ask the user to choose one of the constraints from the explanation. The chosen constraint will then be removed from the constraint problem. Minimality of explanations ensures that no irrelevant constraints are removed. It is important to note that there can be multiple explanations. The number of explanations can even be exponential, but only a linear number of constraints need to be removed from the constraint problem such that the query is no longer entailed by it.

More advanced forms of explanations are needed if the user wants to know the reason underlying a particular constraint in the product model. When a configurator automatically completes a configuration, a user may ask why an expected value has not been chosen by the system. The answer may either be that this value has been inconsistent or that a more preferred value has been chosen instead.

There are other types of questions that we do not consider part of an explanations capability. When testing a configuration against a new product model, a developer can trace the actions performed by the system (firing of constraints, instantiation of components, setting of properties, etc.) and determine performance profiles. As traces and performance profiles deal with the behaviour of a specific configurator, they do not have a declarative nature and are not considered explanations. Furthermore, when finishing an incomplete configuration, a user needs guidance based on the properties of the product model, which is beyond the scope of an explanation.

3 TMS-Based Approaches to Explanation

One mechanism that has been used extensively as a basis for explanations in configurators is the truth maintenance system (TMS). In particular, explanations in the SAP sales configurators have been solely based on this. Two approaches to TMS arise in configuration.

The first and simpler form is a justification-based TMS (JTMS) [5]. It has been in use in SAP configurators since 1992. Thus far, productive configuration solutions by SAP customers almost exclusively make use of explanation based on it.

The second and more complex form is an assumption-based TMS (ATMS) [4]. The approach to using this in configuration was evolved concurrently with that of using a JTMS [13], however, a real need to provide ATMS-based explanations has only recently begun to surface. Limited productive use has been available in the SAP internet configurator (IPC) since 2003.

3.1 Truth Maintenance Systems in Configurators

A TMS is defined by a set of facts $\mathcal{D}$ and a set of justifications $\mathcal{J}$. Each fact is treated by the TMS as an atomic proposition. A justification can be seen as a recorded logical implication between facts $f_1 \cdots f_n \rightarrow f$ where $f_i, f \in \mathcal{D}$.

All reasons for entering a property (fact) into the configuration must be supported by one or more justifications. In particular, each time a constraint in the model is applied, the TMS records this in the form of a justification. Each user input is also recorded as a justification. Thus, the configuration process can be seen as a process of successively communicating justifications to the configurator until a state is reached in which the configuration is considered complete.

An inconsistency is recorded as a justification for a special fact $false$. To handle an inconsistency, one or more of the justifications under user control (e.g. due to foreground constraints) must be retracted or otherwise disabled, which will lead to a previous state. When a justification is retracted all facts that depend solely on it must also be removed. The main reason for using a TMS is actually to support such dependency-directed backtracking, i.e. to rapidly and consistently take back everything that depends on a particular retracted justification.

3.2 The TMS in SAP Sales Configurators

Some amendments have been made to the TMS at SAP. Firstly, facts are not all treated as atomic propositions. They may be partially ordered in a "specialisation relation", with only the most specialised

facts in the configuration being active. This allows facts that represent disjunctions. For example, if it is known that the number of seats of a car must be either six or seven, this can be expressed by a single fact. If subsequently the number of seats is finalised to seven, this information is represented by a more specialised fact. (This feature is not dealt with here, see [14].) Secondly, each justification is additionally qualified by an $owner$. A justification takes the form $f_1 \cdots f_n \xrightarrow{Owner} f$.

$Owner$ may be a constraint, the user, the product model itself, a procedure, or a problem solving heuristic. $Owner$ can have a static text for explanation. This text may contain text variables that can reference the lhs (left-hand side) of the justification. $Owner$ can also distinguish a justification as being retractable by the user or not. Justifications that can be retracted directly by the user (e.g., user input itself and static or dynamic defaults) are called soft justifications. Soft justifications stem from the foreground constraints.

Constraints that involve sums are distinguished from other constraints, both in the model and in the TMS, and are called $aggregation\ checks$. For example, the total electrical power consumed must not exceed that provided. Though typically not explicitly modelled this way for a car, air-conditioning, lights, the engine starter, audio systems, etc., all consume power. Power is provided mainly by the battery (when the engine is off), or the generator (when the engine is running). In this example the total number of all such components is a foreseeable finite number and the relation can be modelled as a classical constraint. In practice, the number of variables involved in a sum often cannot be (practically) bounded in the model. Also, a violation of such a constraint can often be handled by adding further features to the solution, rather than by retracting chosen features. For example, a second battery could be added to the car in case of problematic power supply.

For $aggregation\ checks$ the contributions to the check (the summands) are justified individually. The aggregation itself is computed dynamically at run-time and not justified directly. This has the advantage, from the point of view of explanability, that meaningful static explanation texts can be more easily supplied for the contributions than for the aggregation overall. Also, it avoids unnecessary churning in the TMS processing which would ensue from the frequent replacements of justifications for the entire sum each time a new contribution is added[4].

3.3 JTMS-Based Explanation

In a simple but basic form, any fact including $false$ can be explained by listing all of its justifications and displaying the associated texts (binding any text variables). Note that these justifications can be due to background constraints, which then provide a deeper insight into the reasoning of the TMS. A link to the lhs of each justification is also provided so that the user may request explanations for these facts in turn. For facts representing sums, the summands are identified and their individual explanations are all presented together.

Pursuing explanations on a chain of justifications can be a tedious process. In particular, there is no direct guidance on how to handle an inconsistency. Consequently, some modelers work hard to try to avoid inconsistencies by disallowing any user input that can foreseeably cause problems, i.e they try to reduce the requirement for explanability. Where inconsistencies cannot be avoided, appropriately formulated static explanation texts at the level of $false$ can often provide the necessary guidance for handling the inconsistency without requiring further chaining. Despite these drawbacks this mode of explanation can provide detailed reasons for an inference (such as legal or technical constraints that entail a particular property) that are otherwise not easy to generate. Experience shows that it can be useful to domain experts (e.g. in a B2B scenario).

[4] The latter approach is easier to implement and was used first at SAP.

3.4 ATMS-Based Explanation

In order to remove an inconsistency, or other unwanted fact, from the configuration the user must backward chain on the JTMS explanations until one or more soft justifications are reached that can be retracted. To overcome this deficiency the TMS can be extended to an ATMS. The ATMS supports finding these soft justifications more directly. There are also reasons beyond explanations for utilising an ATMS, but they are not dealt with here.

Using the terminology of the ATMS any soft justification is called an assumption. For each fact f the ATMS calculates an ATMS label $\lambda(f)$ representing a minimal disjunctive normal form of the logical support of the fact in terms of assumptions. Each conjunction in $\lambda(f)$ is called an environment, denoted by ξ. The ATMS label is represented as a list of its environments. Each environment is represented as a list of assumptions. By definition, if $\xi \in \lambda(f)$ then $\xi \vdash f$. If $\lambda(f)$ is empty then f has no support and should not be considered part of the configuration. If $\lambda(f)$ contains the empty environment then f cannot be removed from the configuration by user actions.

Since the label calculation only offers meaningful results if the implications recorded as justifications are logically complete, explanations based on the ATMS presuppose a declarative (constraint-based) modelling paradigm. Procedural inferences will usually not be able to supply logically complete justifications. They can, however, be associated with static texts, thus providing meaningful JTMS-based explanations. Also, facts that represent sums need special consideration. The violation of an $aggregation\ check$ is not necessarily considered an inconsistency if it can be fixed by adding further features.

An environment in the label of $false$ is called a nogood. The ATMS keeps track of the nogoods separately. They and their supersets can then be removed from, or ignored in, the fact labels, because anything that allows deriving $false$ is obviously not of interest. All remaining non-nogood environments in $\lambda(f)$ form the explanation for the presence of f in the configuration.

The direct utility of the ATMS label is that it gives concise information about what needs to be done in order to remove a fact from the configuration. This is particularly useful for $false$, i.e., handling inconsistencies. It can also be used for removing other unwanted features, but this will probably occur less frequently in practice (I don't like air-conditioning in cars, even if I get it for free). A more frequent practical use would be to remove exclusions from the configuration. An exclusion is the direct representation (as a fact) that a property cannot be consistently added to the configuration. Such exclusions are represented in the TMS at SAP either directly (a diesel engine will not fit in the chosen car) or in the form of domain restrictions (the number of seats is less than or equal to five: this excludes six, seven, etc.).

The requirement for this kind of explanation arises more and more in internet scenarios, where less knowledgeable users want to add or remove features without any interest in a deeper explanation of the underlying inferences made by the configurator. Ultimately, some users may require a combination of both explanation functionalities, as provided in the current SAP IPC configurator for some scenarios.

Using an ATMS poses some technical challenges. One such challenge is the potential complexity of the label calculation. Another

challenge is how to present the label in a way that the user can act upon it efficiently. As an example, suppose that an inconsistency needs to be handled and there are several nogoods. To completely handle the inconsistency exactly one soft justification must be withdrawn from each nogood (since a nogood is minimal). Some soft justifications may appear in more than one nogood. Retracting them will thus handle several nogoods at the same time.

The current solution to the complexity problem at SAP is to stop calculation when it becomes seemingly intractable. Explanations through ATMS labels cannot be offered in this case. Experience shows that after optimising the implementation, the label calculation is tractable for most problems encountered in practice so far. After solving problems with an initial implementation, a total label calculation for a complex configuration can, typically, take a few seconds.

The presentation problem is handled in the IPC as follows. A soft justification is presented primarily through the fact that it justifies. A simple heuristic is used to present the list of all facts corresponding to the union of all nogoods. This list is sorted in descending order by the frequency with which a fact occurs in different nogoods. The user is asked to choose one fact he is willing to forego. Each fact represents one or more soft justifications that will be retracted if the corresponding fact is chosen. Facts towards the top of the list are linked to soft justifications that are involved in more nogoods. After the user has made a choice and the corresponding soft justifications are retracted, the resulting smaller list is again displayed for a further choice if needed. Both the label calculation itself and its presentation for explanation are current topics of further improvement.

4 The QuickXplain Approach

As explained in Section 2, explanations of failure help the user to identify problematic constraints that need to be removed or modified in order to restore consistency. It is therefore important that explanations contain only relevant elements and are minimal with respect to set inclusion. QuickXplain [16] is a method for computing minimal explanations for arbitrary constraint solvers. We discuss some of the motivations for QuickXplain.

4.1 Motivations for QuickXplain

A TMS tracks the precise inferences of a constraint solver and records them as justifications. The explanations can then be computed from the justifications as shown in the previous section. Coupling an existing constraint solver with a TMS is intrusive and requires some modifications to the solver code. Furthermore, many advanced constraint solvers exploit efficient algorithms for constraint propagation (such as the famous `alldifferent` constraint [27]). It is difficult to track the deductions made by those algorithms without slowing them down. Furthermore, the resulting justifications may be very large even if the final explanations are small. Moreover, the explanations obtained from those justification networks can be far from being minimal. We show this for a simple example comprising a sum-constraint, four requirements and the binary variables x_1, x_2, x_3:

$$c : y = 500 \cdot x_1 + 400 \cdot x_2 + 2600 \cdot x_3,$$
$$r_1 : x_1 = 1, \quad r_2 : x_2 = 1, \quad r_3 : x_3 = 1, \quad r_4 : y \leq 3000.$$

Each propagation step of this problem is recorded by a justification. A set of justifications that allows us to derive a failure represents an inconsistency proof. The following is an inconsistency proof for the example, showing that the requirements r_1, r_2, r_3, r_4 are in conflict:

$$r_1, c \rightarrow y \geq 500$$
$$r_2, y \geq 500, c \rightarrow y \geq 900$$
$$r_3, y \geq 900, c \rightarrow y \geq 3500$$
$$r_4, y \geq 3500 \rightarrow \bot$$

However, there is another inconsistency proof for this problem which leads to a smaller explanation of the failure, which is minimal:

$$r_1, c \rightarrow y \geq 500$$
$$r_3, y \geq 500, c \rightarrow y \geq 3100$$
$$r_4, y \geq 3100 \rightarrow \bot$$

The computation of minimal explanations of failure thus requires the exploration of multiple inconsistency proofs that are obtained by different subsets of the constraints. The ATMS achieves this by exploring the logical consequences of all (relevant) subsets and computing all minimal explanations.

The number of subsets to be investigated can be reduced significantly if a single explanation is required. Furthermore, it is not necessary to determine those subsets by propagating sub-explanations over a justification network. It is simply possible to choose those subsets outside the problem solver. The result is a non-intrusive explanation method that can be used with any problem solver without modification. It can thus be used with advanced constraint solvers based on global constraints. It can also be used to reduce non-minimal explanations produced by TMS-based solvers such as modern SAT solvers. Finally, it also works for solvers that do not exploit logical inference, but optimisation for proving inconsistency, e.g. LP solvers.

QuickXplain was first developed for ILOG's configuration tools and provides a critical feature for configuration applications. It is also included in ILOG's constraint programming tools. Furthermore, it has been applied to very diverse problems such as Benders decomposition for resource allocation, explanations for semantic web, and validation of business rules where it is used to find rule conditions that ensure an ill-formulated rule is never applicable.

4.2 Principles of QuickXplain

QuickXplain isolates a conflict by successively removing constraints and by checking the consistency of the remaining constraints. If the remaining constraints are inconsistent, the problem has been reduced in size. If not, then some of the removed constraints do necessarily belong to the conflict. Consider a similar example as in the last section where the background B contains a sum-constraint $\sum_{i=1}^{n} k_i \cdot x_i \leq u$ and the foreground contains the requirements $r_i : x_i = 1$. We first consider a simple problem where $n = 8$, $k_i = 2^{i-1}$, and $u = 9$. This problem is inconsistent and we use QuickXplain to determine a minimal explanation of failure.

QuickXplain is supplied with a tuple (B, C) consisting of the original background B and the original foreground C, which we rename into $C_{1,8}$. QuickXplain will map this explanation problem to a sequence of subproblems by moving constraints from its current foreground to its current background. Firstly, the algorithm checks whether the background itself is consistent, which is the case since the sum-constraint can be satisfied by, for example, setting all x_i to 0. It then recursively divides the explanation problem into subproblems of the same size. The foreground $C_{1,8}$ will be split into two subsets $C_{1,4} := \{r_1, \ldots, r_4\}$ and $C_{5,8} := \{r_5, \ldots, r_8\}$. The algorithms first seeks for conflict elements in $C_{5,8}$ while keeping $C_{1,4}$ in the background (i.e., all constraints in $C_{1,4}$ are active, but they will

not be removed while solving the second problem). Hence, it seeks an explanation for $P_{5,8} := (B \cup C_{1,4}, C_{5,8})$. It again checks for the consistency of the new background $B \cup C_{1,4}$. However, this problem has no solution as $\sum_{i=1}^{4} 2^{i-1}$ is equal to 15 and exceeds 9. Due to this failure, the empty set $X_{5,8} := \emptyset$ is a conflict of the second subproblem $P_{5,8}$, meaning that the set of conflicting constraints has been reduced to $C_{1,4}$.

The algorithm now solves the first subproblem, namely $P_{1,4} := (B, C_{1,4})$. It already knows that the background B is consistent and therefore splits $C_{1,4}$ into $C_{1,2} := \{r_1, \ldots, r_2\}$ and $C_{3,4} := \{r_3, \ldots, r_4\}$. Again, it seeks culprits among $C_{3,4}$ while moving $C_{1,2}$ into the background. We thus obtain the problem $P_4 := (B \cup C_{1,2}, C_{3,4})$. The background $B \cup C_{1,2}$ is consistent, and at least one constraint of $C_{3,4}$ is needed for the failure. QuickXplain splits this set into $C_3 := \{r_3\}$ and $C_4 := \{r_4\}$. The background of $P_4 := (B \cup C_{1,2} \cup C_3, C_4)$ contains r_1, r_2, r_3 and is consistent. Hence, C_4 must contain an element of the conflict. Since it is a singleton, $X_4 := \{r_4\}$ is a minimal conflict of the subproblem P_4.

This conflict is added to the background of the subproblem P_3 which thus has the form $(B \cup C_{1,2} \cup X_4, C_3)$. The background of P_3 contains r_1, r_2, r_4 and is not consistent. Thus, the empty set is a minimal conflict of P_3. The minimal conflict of $P_{3,4}$ is obtained by merging the minimal conflicts of its subproblems P_3 and P_4, which results in $X_{3,4} := \{r_4\}$.

This conflict is added to the background of $P_{1,2} := (B \cup X_{3,4}, C_{1,2})$. The background of $P_{1,2}$ contains r_4 and is consistent. We thus need an element of $C_{1,2}$, which is split into $C_1 := \{r_1\}$ and $C_2 := \{r_2\}$. The background of the second subproblem $(B \cup X_{3,4} \cup C_1, C_2)$ contains r_4, r_1 and is consistent. Since C_2 is a singleton, $X_2 := \{r_2\}$ is the minimal conflict of this problem. Finally, the algorithm seeks an explanation of $P_1 := (B \cup X_{3,4} \cup X_2, C_1)$. The background now contains r_4, r_2, which is consistent, meaning that the explanation is $X_1 := \{r_1\}$. The explanation $X_{1,2}$ of $P_{1,2}$ is then the union of X_1 and X_2, and the explanation $X_{1,4}$ is equal to $X_{1,2} \cup X_{3,4} = \{r_1, r_2, r_4\}$. Since $X_{5,8}$ was empty, $X_{1,4}$ is the minimal conflict of the initial problem.

4.3 Usability Issues of QuickXplain

The efficiency of QuickXplain depends of the number of consistency checks and the costs of consistency checks. If a consistency check fails, the current explanation problem is reduced in size and a whole block of constraints is removed from the problem. If the culprits are grouped together, then the blocks become very large and QuickXplain is very effective. However, even if k culprits are equally distributed, we still get blocks of size $O(n/k)$ and QuickXplain can still bring significant savings over naive methods. QuickXplain needs between $\log_2 \frac{n}{k} + 2k$ and $k \cdot \log_2 \frac{n}{k} + 2k$ consistency checks. It is thus logarithmic in n iff k is small compared to n.

Although configuration problems can be large in size, the number n of foreground elements will usually be quite small for interactive configurators. Furthermore, interactive configurators usually use constraint propagation to derive required features and to eliminate the possible features. In this case, QuickXplain will use the same propagation procedure for the consistency checks. As constraint propagation is incremental with respect to the addition of constraints, QuickXplain can perform very efficiently for interactive configurators.

For model debugging, the number n of foreground elements can be large. QuickXplain will still be beneficial if the conflicts are small in size, which is often the case for modelling errors. If conflicts are large in size, it might be sufficient to stop QuickXplain after it has determined the m last elements of a conflict. Model debugging may require the usage of a search procedure to check the consistency of the constraints associated with a component type. This is feasible if search can be limited to critical variables.

QuickXplain determines a single explanation at a time and needs to be reapplied if the removal of a constraint does not restore consistency. This sequential approach will determine at most n explanations, even if the initial number of explanations is exponential. However, the user may want to control which explanation is found and impose a preference order on the foreground constraint. This order uniquely defines a preferred explanation and QuickXplain determines exactly this preferred explanation.

5 Advanced Issues

In this section we discuss a number of topics that can be regarded as extensions to, or advanced aspects of, explanation generation. We limit our attention to consistency- and constraint-based approaches.

Explanations are closely related to the Partial Constraint Satisfaction (PCSP) framework [11]. The PCSP framework is useful for reasoning about problems that do not admit solutions. Of course, in reality such a situation is unsatisfactory. Therefore, based on the PCSP framework we can define a number of ways of relaxing a CSP so that a solution can be found, e.g. by allowing some constraints to be violated. Many techniques have been proposed that reason about over-constrained systems of constraints [3, 26]. Many of these techniques can be seen as examples of explanation generators, although they have not been designed for that purpose. The notion of PCSP has also be used as the underlying framework for computing a specific kind of explanation know as a tradeoff [9]. Tradeoffs can be regarded as an explanation of how to overcome an inconsistent set of constraints, while directing a user towards a preferred solution.

Many other generalisations of the basic constraint satisfaction paradigm have been proposed specifically with configuration in mind. For example, Dynamic CSP [22] is an extension of the framework in which the existence of particular variables and constraints in a configuration are conditional on particular assignments to the variables. For example, air-conditioning systems may only be available in luxury models of cars. In this case, if the type of car is represented by a variable, assigning this variable to the value luxury would introduce another variable called air-conditioner, for which the user could select from a set of alternative systems. Explanation generation techniques that can support the dynamic aspects of configuration are very important. A number of such approaches have been proposed, essentially based on forms of truth maintenance [2].

A generalisation of the Dynamic CSP framework is the notion of Composite CSP [29]. This extension was motivated by the observation that configuration problems cannot be adequately modelled using "flat" models. Instead their structure should be modelled directly. Such structure often takes a hierarchical form in which the overall product is composed of subsystems, each of which is composed of sub-subsystems and so on. From the perspective of explanation generation, not only must one be able to explain at a system/subsystem level, but must also be able to generate explanations based on the internal structure of these systems. A number of techniques for generating explanations of structured systems have been developed in the model-based reasoning community [20, 21].

An important issue that must be considered when generating explanations is the notion of "quality", i.e. how good are the explanations from the perspective of the user. This is a very difficult concept to define. Indeed, it has been the topic of a large body of work in

the cognitive science and artificial intelligence communities [19, 31]. Recent work in configuration has considered the notion of *spurious* and *well-founded* explanations, how they arise, and how they can be avoided [12]. A spurious explanation is essentially one that directs a user towards one or more undesired solutions. A spurious explanation can arise when a user asks a configurator to justify a proposed solution in terms of his set of requirements. Since minimal explanations focus on a subset of the user's requirements, explanations can be given that unsoundly explain the proposed solution.

The techniques used to compute well-founded explanations also complement more abstract feature-based explanations. For example, rather than generating explanations in terms of product features, a set of features can be grouped together to form a more higher level concept in terms of product function that a user might find more natural[5]. For example, it is often easier for users to talk about whether they can use a digital camera to take pictures at night, than define the set of technical specifications that enable this function. Abstraction techniques have been used successfully within the constraint satisfaction paradigm [10]. Combining abstraction techniques, for example based on interchangeable values [7], with explanation is very promising.

There is a considerable amount of work on explanation in constraint satisfaction. Many researchers have addressed the generation of explanations in interactive constraint-based systems [1, 2, 8, 24]. There has also been work on explanations that focus on explaining why a particular solution to a problem exists [18, 25, 30].

As highlighted earlier in this paper, one possible type of user in configuration are those who construct the knowledge bases upon which configurators are built. The typical form of explanation required by these users is based on concepts from model debugging [6]. The complexity of industrial configuration knowledge-bases present a significant challenge for debugging. Consistency-based diagnosis has been proposed to generate explanations that identify the sources of bugs so that they can be successfully eliminated. Traditional approaches to diagnosis, such as Reiter's hitting set algorithm [28] can be used to compute the required explanation describing how the bug can be removed.

While traditional approaches to explanation focus on finding single explanations, configurators can be extended to help users find an explanation that they find satisfactory. Given an initial explanation, a user might find it unhelpful and may request that an explanation be presented which is quite different from the initial one. For example, the former explanation might be in terms of technical features while the second explanation might be in terms of cost. Alternatively, the user might seek to refine an explanation by finding an alternative, but similar, one. This interaction can be readily supported using a known framework for reasoning about similarity and diversity problems in constraint satisfaction [15].

6 Conclusion

We have discussed the scope of explanations; the requirements of different classes of user from the perspective of explanation; and discussed two of the standard approaches to explanation in industrial configuration: TMS and QuickXplain. Finally, we summarised some advanced issues related to explanations in configuration.

REFERENCES

[1] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis, 'Consistency restoration and explanations in dynamic CSPs application to configuration.', *Artificial Intelligence*, **135**(1-2), 199–234, (2002).

[2] James Bowen, 'Using dependency records to generate design coordination advice in a constraint-based approach to concurrent engineering', *Computers in Industry*, **22**(1), 191–199, (1997).

[3] Simon de Givry, Javier Larrosa, Pedro Meseguer, and Thomas Schiex, 'Solving Max-SAT as weighted CSP.', in *CP*, pp. 363–376, (2003).

[4] Johan de Kleer, 'An assumption–based truth maintenance system', *Artificial Intelligence*, **28**, 127–162, (1986).

[5] Jon Doyle, 'A truth maintenance system', *Artificial Intelligence*, **12**, 231–272, (1979).

[6] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner, 'Consistency-based diagnosis of configuration knowledge bases.', *Artif. Intell.*, **152**(2), 213–234, (2004).

[7] Eugene C. Freuder, 'Eliminating interchangeable values in constraint satisfaction problems.', in *AAAI*, pp. 227–233, (1991).

[8] Eugene C. Freuder, Chavalit Likitvivatanavong, Manuela Moretti, Francesca Rossi, and Richard J. Wallace, 'Computing explanations and implications in preference-based configurators', in *Recent Advances in Constraints*, LNAI 2627, pp. 76–92, (2003).

[9] Eugene C. Freuder and Barry O'Sullivan, 'Generating tradeoffs for interactive constraint-based configuration.', in *CP*, pp. 590–594, (2001).

[10] Eugene C. Freuder and Daniel Sabin, 'Interchangeability supports abstraction and reformulation for multi-dimensional constraint satisfaction.', in *AAAI/IAAI*, pp. 191–196, (1997).

[11] Eugene C. Freuder and Richard J. Wallace, 'Partial constraint satisfaction.', *Artif. Intell.*, **58**(1-3), 21–70, (1992).

[12] Gerhard Friedrich, 'Elimination of spurious explanations', in *Sixteenth European Conference on Artificial Intelligence*, pp. 813–817, (2004).

[13] Albert Haag, 'Konzepte zur praktischen handhabbarkeit einer atms basierten problemloesung.', in *Das Plakon Buch*, eds., R. Cunis, A. Günter, and H. Strecker, pp. 212–237. Springer Verlag, (1991).

[14] Albert Haag, 'Sales configuration in business processes', *IEEE Intelligent Systems*, **13**(4), 78–85, (1998).

[15] Emmanuel Hebrard, Brahim Hnich, Barry O'Sullivan, and Toby Walsh, 'Finding diverse and similar solutions in constraint programming.', in *AAAI*, pp. 372–377, (2005).

[16] Ulrich Junker, 'QuickXplain: preferred explanations and relaxations for over-constrained problems', in *Proceedings of AAAI*, pp. 167–172, (2004).

[17] Ulrich Junker, 'Configuration', in *Handbook of Constraint Programming*, chapter 24, Elsevier, (2006).

[18] Narendra Jussien and Vincent Barichard, 'The PaLM system: explanation-based constraint programming', in *Proceedings of CP-2000 TRICS Workshop*, pp. 118–133, (2000).

[19] David Leake, *Evaluating explanations: A content theory*, Lawrence Erlbaum Associates, 1992.

[20] Jakob Mauss and Mugur M. Tatar, 'Computing minimal conflicts for rich constraint languages.', in *ECAI*, pp. 151–155, (2002).

[21] Wolfgang Mayer and Markus Stumptner, 'Model-based debugging using multiple abstract models', *CoRR*, **cs.SE/0309030**, (2003).

[22] Sanjay Mittal and Brian Falkenhainer, 'Dynamic constraint satisfaction problems', in *Proceedings of AAAI*, pp. 25–32, (July–August 1990).

[23] Sanjay Mittal and Felix Frayman, 'Towards a generic model of configuration tasks', in *Proceedings of IJCAI*, pp. 1395–1401, (1989).

[24] Barry O'Callaghan, Barry O'Sullivan, and Eugene C. Freuder, 'Generating corrective explanations for interactive constraint satisfaction', in *Proceedings of CP*, pp. 445–459, (2005).

[25] Samir Ouis, Narendra Jussien, and Patrice Boizumault, 'COINS: a constraint-based interactive solving system', in *ICLP-2002 Workshop on Logic Programming Environments*, pp. 31 – 46, (2002).

[26] Thierry Petit, Jean-Charles Régin, and Christian Bessière, 'Specific filtering algorithms for over-constrained problems.', in *CP*, pp. 451–463, (2001).

[27] J-C. Regin, 'A filtering algorithm for constraints of difference in CSPs', in *Proc. of AAAI*, pp. 362–367, (1994).

[28] Raymond Reiter, 'A theory of diagnosis from first principles', *Artificial Intelligence*, **32**(1), 57–95, (1987).

[29] Daniel Sabin and Eugene C. Freuder, 'Configuration as composite constraint satisfaction', in *Fall Symp. on Configuration*, pp. 28–36, (1996).

[30] Mohammed H. Sqalli and Eugene C. Freuder, 'Inference-based constraint satisfaction supports explanation', in *Proceedings of AAAI*, pp. 318–325, (1996).

[31] Richard J. Wallace and Eugene C. Freuder, 'Explanations for whom?', in *Proc. of UICS01*, pp. 119 – 129, (2001).

[5] See `http://www.activebuyersguide.com`.

Solver Framework for Conditional Constraint Satisfaction Problems

Esther Gelle[1] and **Mihaela Sabin**[2]

Abstract. Real world tasks with dynamic behavior, such as configuration, design, planning, or hardware test generation, have been modeled with an extension of standard constraint satisfaction, Conditional Constraint Satisfaction (CondCSPs). CondCSPs capture problem change at solving time by conditionally identifying those variables and constraints that are relevant to final solutions. In this paper we present a CondCSP solver that includes two algorithms for direct and reformulation solving of CondCSPs. Our experimental analysis using randomly generated CondCSPs shows that reformulation solving in conjunction with forward checking performs better on problems with larger solution spaces than the direct solving methods.

1 Introduction

A conditional constraint satisfaction problem (CondCSP) extends the standard CSP with a condition-based component that models problem change by allowing for "on-the-fly" selection of subsets of variables that participate in problem solutions. The formalism, conditional CSP, has been introduced by [7] and further developed by [9]. The original application domain is product configuration, in which a changing rather than fixed number of components are part of final solutions. More recently, constraint satisfaction has been adapted to the planning domain [6], [1]. In [1] the following advantages of CSP over SAT are stated: a) CSP encodings tend to be less memory-savvy b) CSP encodings exhibit more structure c) this structure is exploited by enforcing partial consistency using standard CSP search techniques. Hardware test generation is another example of applying a slightly modified variant of CondCSP, called Activity CSP, where clustering and disjunction of problem activity are represented explicitly [5]. The Activity CSP reformulates condition-based constraints into standard constraints by means of additional variables that control variable activity. This reformulation is more efficiently solved by enforcing an early constraint propagation method in which constraints are invoked based on assumptions that some variables are active.

Each application domain has produced either specialized algorithms for solving CondCSPs or reformulated CondCSPs into standard CSPs. The CondCSP class lacks systematic findings with regard to how algorithm efficiency correlates with problem topology, such as density, satisfiability, and conditionality and also by comparing the algorithms' relative performance. This challenge is compounded by an almost inexistent library of CondCSP benchmark problems.

To address these challenges, we have developed a CondCSP solver that includes two algorithms, each of which has initially been introduced and evaluated separately. One algorithm has direct solving methods that adapt standard consistency checking, such as forward checking and maintaining arc consistency, to the special constraints that enforce conditionality in a CondCSP [8], [10]. The other algorithm reformulates the original problem into intermediate CondCSPs with incrementally lesser conditionality as they are ultimately transformed into standard CSPs. Standard consistency checking is interleaved with problem reformulation to eliminate inconsistent subproblems and solve the resulting standard CSPs [4], [2]. To overcome the lack of publicly known benchmark problems, we have used random CondCSPs and designed test suites for both direct and reformulation solving algorithms. Initial work has been described in [3]. The extended evaluation analysis in this paper shows that there is not one winner. Reformulation solving in conjunction with forward checking has the advantage of pruning whole subproblems, and thus it copes better with larger-size problems (large solution spaces in the tens of thousands) than the direct solving methods. Secondly, direct solving in conjunction with maintaining arc consistency is always preferred over direct solving using forward checking, and is more efficient on large, overconstrained problems.

[1] ABB Switzerland Ltd, Corporate Research, CH-5405 Baden, Switzerland
[2] Department of Mathematics and Computer Science, Rivier College, 420 Main Street, Nashua, NH 03060

2 A Conditional CSP Example

A CondCSP, $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$, has a set of variables, $\mathcal{V} = \{v_1, \ldots, v_n\}$, which, if active, can take on discrete values from their corresponding finite domains $\mathcal{D} = \{D_{v_1}, \ldots, D_{v_n}\}$; a non-empty set of initially active variables, called initial variables, $\mathcal{V}_I$, $\mathcal{V}_I \subseteq \mathcal{V}$; a set of compatibility constraints, $\mathcal{C}_C$; and a set of activity constraints, $\mathcal{C}_A$. All sets are finite. We use as guiding example a simplified configuration problem for industrial mixers [2] with requirements shown in Figure 1. The task of configuration is to assign values to all selected components in such a way that all requirements are met. We model the mixer configuration task as a CondCSP as follows (Figure 2). Configuration components and their values correspond to the problem's *variables* $\mathcal{V} = \{Type, Volume, Process, Cooler, Condenser\}$ and *domains of values* $\mathcal{D}$. The required components are called *initial variables* $\mathcal{V}_I = \{Type, Volume, Process\}$. They are initially *active* or *included* in the problem search space and present at any time during search. Optional components have their *activity status* initially *undefined*; in the example $Cooler$ and $Condenser$ have their activity status initially undefined. Requirements for selecting these optional components are modeled through *activity constraints* $\mathcal{C}_A = \{a_1, a_2, a_3, a_4\}$. These constraints extend the initial variable set according to certain *activation conditions* and thus change the activation status of the variables representing the optional components to indicate either *inclusion* from solutions as in a_1, a_2, and a_4 or *exclusion* as in a_3. Requirements of component compatibility correspond to *compatibility*

constraints $\mathcal{C}_C = \{c_1, c_2\}$, which restrict the combinations of allowed values assigned to selected components.

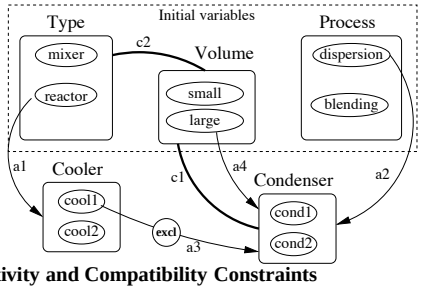
Required components and their values
- mixer's vessel type is mixer or reactor
- vessel's volume is small or large
- mixing process is dispersion or blending

Optional components and their values
- cooler's type is cool1 or cool2
- condenser's type is cond1 or cond2

Configuration requirements of component compatibility
1. small volume is incompatible with condenser's cond1
2. mixer and reactor type is compatible with small volume

Configuration requirements for selecting optional components
1. reactor type includes the cooler option
2. dispersion process includes the condenser option
3. cool1 cooler excludes the condenser option
4. large volume includes the condenser option

Figure 1. Example of a simplified industrial mixer configuration task



Activity and Compatibility Constraints

a_1 : $Type = reactor \xrightarrow{incl} Cooler$

a_2 : $Process = dispersion \xrightarrow{incl} Condenser$

a_3 : $Cooler = cool1 \xrightarrow{excl} Condenser$

a_4 : $Volume = large \xrightarrow{incl} Condenser$

c_1^{dis} : $\{(small, cool1)\}$

c_2 $\{(mixer, small)(reactor, small)\}$

Solution Set

Type	Process	Volume	Cooler	Condenser
reactor	dispersion	small	cool2	cond2
reactor	blending	small	cool1	EXCL
reactor	blending	small	cool2	UNDEF
mixer	dispersion	small	UNDEF	cond2
mixer	blending	small	UNDEF	UNDEF

Figure 2. CondCSP of the mixer configuration example

A compatibility constraint c is consistent with an instantiation $\mathcal{I}$ of the constraint variables iff either not all constraint variables are active, or constraint variables are active and $\mathcal{I}$ satisfies c. For example, the instantiation $Type = mixer$, $Volume = large$, and $Process = blending$ trivially satisfies c_1 since the variable $Condenser$ has the activity status undefined.

An inclusion activity constraint, $a : a_{cond} \xrightarrow{incl} v_t$, has an activation condition, a_{cond}, which is a regular constraint defined on a set of condition variables, and a target variable, v_t. The activity constraint a is consistent with an instantiation $\mathcal{I}$ of the condition variables of a_{cond} iff either (1) not all condition variables are active or a_{cond} is inconsistent with $\mathcal{I}$, or (2) all condition variables are active, $\mathcal{I}$ satisfies a_{cond}, and v_t is active. In our example, the instantiation $Type = reactor$ makes $Cooler$ active according to a_1. The condition variable $Type$ is initially active and the instantiation satisfies the activation condition, thus the a_1 is consistent with $Type = reactor$.

Given an exclusion activity constraint, $a : a_{cond} \xrightarrow{excl} v_t$, a is consistent with an instantiation $\mathcal{I}$ of the condition variables a_{cond} iff (1) either not all condition variables are active or a_{cond} is inconsistent with $\mathcal{I}$, or (2) all condition variables are active, $\mathcal{I}$ satisfies a_{cond}, and v_t is not active. The instantiation $Type = reactor$, $Process = blending$, $Volume = small$, and $Cooler = cool1$ excludes $Condenser$, since condition variable $Cooler$ is active (a_1 is satisfied) and the instantiation satisfies the activation condition.

A solution to a CondCSP $\mathcal{P}$ is an instantiation of a set of active variables such that all compatibility and activity constraints are satisfied. We are interested in generating all solutions to the example problem (listed in Figure 2).

3 Solving Conditional CSPs

The large collection of thoroughly tested algorithms for solving standard CSPs is in stark contrast with the few existing algorithms for solving CondCSPs. The first complete description of CondCSP backtrack search [4] solves a partially reformulated CondCSP, in which activity constraints of exclusion are rewritten as compatibility constraints. [8] proposed CondCSP analogs to CSP backtrack ($CondBt$), forward checking ($CondFc$), and maintaining arc consistency ($CondMac$) search algorithms. $CondMac$ interleaves backtrack search with maintaining arc consistency, which is adapted to propagate consistency checking on both compatibility and activity constraints in the original CondCSP. Experimental evaluation on random CondCSPs [11] shows up to two orders of magnitude of efficiency improvement over plain backtrack search.

A different solving approach is to successively process activation conditions of inclusion into an equivalent reformulation [4], [2]. The reformulation algorithm, Gt, generates a tree whose internal nodes are CondCSPs and the leaves are standard CSPs. Gt reformulates inclusion activity constraints into compatibility constraints by creating intermediate CondCSPs with lesser conditionality until the leaves level is reached. Standard consistency checking is interleaved with tree generation to eliminate inconsistent subproblems and to solve in the end the resulting standard CSPs. Experimental evaluation results of Gt using forward checking were reported for solving a bridge design problem.

For the purpose of examining these algorithms' relative performance we have integrated both implementations within the same solver framework (written in C++).

3.1 Direct Solving: $CondMac$ Algorithm

Arc consistency over compatibility constraints is handled by a regular, $AC4$-based Mac method, which we call $MacCompatibility$ to distinguish it from consistency checking over activity constraints. Activity constraints are checked with the $MacActivity$ method (Algorithm 1). If activity constraints relevant to the variable assignment, $var = value$, are satisfied, variables newly included and excluded by the checked activity constraints are made arc consistent over activity and compatibility constraints.

The purpose of $MacActivity$ is to collect newly included and excluded target variables and to use $MacNewvar$ method (Algorithm 2) to make these variables arc consistent. $MacNewvar$ has two cases, depending on the activity status of the $newvar$ as either included or excluded. The excluded activity status might make values at future condition variables inconsistent. This is the case when those condition variables are part of inclusion activity constraints that have not been processed yet, but target $newvar$. The second case is when

Algorithm 1 *Maintaining arc consistency over activity constraints.*

boolean $MacActivity(var, value, Agenda, UndoVals, UndoAct)$ {
  $A \leftarrow$ activity constraints whose conditions involve var
  for each ($a \in A$) {
    if ($value$ is **not** consistent with a's condition)
      return true //no effect on var's activity status
    else {
      $target \leftarrow$ target variable of a
      $action \leftarrow$ activity performed by a
      if ($action$ includes $target$) {
        if ($target$ has already been excluded)
          return false //conflicting activity constraints
        if ($target$ is newly included) {
          $Agenda \leftarrow Agenda \cup target$
          $NewVariables \leftarrow NewVariables \cup target$}}
      else { // $action$ excludes $target$
        if ($target$ has already been included)
          return false //conflicting activity constraints
        if ($target$ is newly excluded)
          $NewVariables \leftarrow NewVariables \cup target$ }
      $UndoAct \leftarrow UndoAct \cup a$ }
  }//end of A list
  for each ($newvar \in NewVariables$) {
    $LocalUndoValues \leftarrow \emptyset$
    macNewResult $\leftarrow MacNewvar(newvar, LocalUndoValues)$
    $UndoVals \leftarrow UndoVals \cup LocalUndoValues$
    if (macNewResult is **false**)
      return false
  }//end of $NewVariables$ list
  return true
}//end $MacActivity()$

$newvar$ is included. Local consistency considers both compatibility and activity constraints. $newvar$ has to be arc consistent with all the other active variables with which it shares compatibility constraints.

Algorithm 2 *Maintain arc consistency with a newly included or excluded variable.*

boolean $MacNewvar(newvar, Agenda, UndoVals)$ {
  if ($newvar$ is newly excluded)
    $InclTarget \leftarrow$ activity constraints that include $newvar$ as target
    for each ($it \in InclTarget$, where $it : CondVar = val \xrightarrow{incl} newvar$
        and $CondVar$ is on the $Agenda$)
      if ($removeValue(val, CondVar, UndoVals)$ is **false**)
        return false
  else // newvar is newly included
    $C \leftarrow$ compatibility constraints on $(newvar, othervar)$
      with $othervar$ active
    for each ($c \in C$)
      $makeOneAC(c, newvar, othervar, UndoVals)$
    $ExclTarget \leftarrow$ activity constraints which exclude target $newvar$
    for each ($et \in ExclTarget$
        where $et : CondVar = val \xrightarrow{excl} newvar$
        and $CondVar$ is on the $Agenda$)
      if ($removeValue(val, CondVar, UndoVals)$ is **false**)
        return false
    $SourceAct \leftarrow$ activity constraints whose source is $newvar$
    for each ($sa \in SourceAct$, such that
        either sa includes an already excluded target
        or sa excludes an already included target in the $Agenda$)
      if ($removeValue(condition, newvar, UndoVals)$ is **false**)
        return false
  return $MacCompatibility(UndoValues, Agenda)$
}//end $MacNewvar()$

The included variable status makes possible two roles the variable might play in activity constraints that have not been processed yet: as a condition variable or a target variable. As a condition variable, $newvar$ can participate in either (1) an inclusion activity constraint that targets an excluded variable, or in (2) an exclusion activity constraint that targets an included variable. In either situation, $newvar$'s condition value is inconsistent with these activity constraints. As a target variable, since $newvar$ status is included, it can render inconsistent condition values of exclusion activity constraints. Finally, all value removals saved in the $UndoValues$ structure are propagated with $MacCompatibility$. If the propagation is successful, $newvar$ is made consistent with the problem variables on both types of constraints.

Algorithm 3 *CondCSP direct solving.*

boolean $CondMac(Agenda)$ {
  if ($Agenda$ is empty)
    return true
  $var \leftarrow$ select variable and remove from $Agenda$
  $value \leftarrow$ select value from domain of var **and** instantiate var
  $UndoActivity \leftarrow \emptyset$
  remove all values from domain of var except $value$
  $UndoVals \leftarrow$ pairs (var, rv), where rv is var's removed value
  if ($MacCompatibility(UndoVals, Agenda)$ **and**
    $MacActivity(var, value, Agenda, UndoVals, UndoActivity)$)
      $CondMac(Agenda)$
  restore all removed values saved in $UndoVals$
  reset variable activity status as saved in $UndoActivity$
  uninstantiate var **and** put it back into the $Agenda$
  remove $value$ from domain of var
  $UndoValues \leftarrow \{(var, value)\}$
  if (domain of var is empty) $backtrackSearch \leftarrow$ **false**
  else if (**not** ($MacCompatibility(UndoValues, Agenda)$))
    $backtrackSearch \leftarrow$ **false**
  else
    $backtrackSearch \leftarrow CondMac(Agenda)$
  reset variable activity status as saved in $UndoActivity$
  restore all removed values saved in $UndoVals$
  return $backtrackSearch$
}//end $CondMac()$

$CondMac$ (Algorithm3) selects a variable var, instantiates it with a $value$, removes all the other values from the domain of var, and saves them in the $UndoValues$ list. Compatibility constraints are propagated with $MacCompatibility$ to reestablish arc consistency among var and future variables. Upon successful return of this methods, $MacActivity$ is called to check activity constraints and make new variables added to the search space arc consistent with regard to both types of constraints. If no activity constraint fails and the new variables do not cause wiping off future variable domains, $CondMac$ is called recursively to find value assignments to the rest of the problem variables. In case of failure in trying $value$ for var or searching for the next solution, variable status information and changes recorded in $UndoValues$ and $UndoActivity$ have to be undone. The current instantiation $value$ is marked as tried and removed from the domain of var. If other values are left in var's domain, $value$ removal is propagated first with $MacCompatibility$ to check whether future variables remain arc consistent with the rest of values at var. If that is the case, a recursive call to $CondMac$ continues the search to find all solutions.

3.2 Reformulation Solving: $GtMac$ Algorithm

$GtMac$ processes the next activity constraint with a condition defined on an active variable. The selected activity constraint is used to split the search space into two branches. The first branch is generated with the $WithCondition$ method, and contains solutions that satisfy the activation condition $var = val$. The corresponding generated CSP has all currently active variables, the target variable $newvar$, and compatibility constraints defined on them. The second branch is generated with $WithNegCondition$ and contains solutions that satisfy the negated activation condition $\neg(var = val)$. The corresponding generated CSP has all currently active variables and compatibility constraints defined on them. Both search spaces

Algorithm 4 *$GtMac$ reformulation solving.*

$GtMac(InclV, InclC, CC, AC)$ {
  $ac \leftarrow$ activity constraint in AC whose condition on $InclV$
  if ac not found
    $P_0 \leftarrow \{InclV, InclC\}$
    solve P_0
  else
    remove ac from AC
    $P_1 \leftarrow WithCondition(ac, InclV, InclC, CC)$
    if $(P_1 \neq \emptyset)$ $GtMac(InclV_{P1}, InclC_{P1}, CC, AC)$
    $P_2 \leftarrow WithNegCondition(ac, InclV, InclC)$
    if $(P_2 \neq \emptyset)$ $GtMac(InclV_{P2}, InclC_{P2}, CC, AC)$
}//end $GtMac()$

$WithCondition(ac, InclV, InclC, CC)$ {
  $newcon \leftarrow ac$'s condition
  $newvar \leftarrow ac$'s target variable
  $InclV \leftarrow InclV \cup newvar$
  $newconstraints \leftarrow$ compatibility constraints on $InclV \subseteq CC$
  $InclC \leftarrow InclC \cup newcon \cup newconstraints$
  if $CheckLocalConsistency(InclV, InclC)$
    $generatedCSP \leftarrow \{InclV, InclC\}$
  else $generatedCSP \leftarrow \emptyset$
  return $generatedCSP$
}//$WithCondition()$

$WithNegCondition(ac, InclV, InclC)$ {
  $newnegcon \leftarrow ac$'s negated condition
  $InclC \leftarrow InclC \cup newnegcon$
  if $CheckLocalConsistency(InclV, InclC)$
    $generatedCSP \leftarrow \{InclV, InclC\}$
  else $generatedCSP \leftarrow \emptyset$
  return $generatedCSP$
}//$WithNegCondition()$

are made locally arc consistent with $MacCompatibility$. If all activity constraints are exhausted and the current search space is locally arc consistent, $MacCompatibility$ is applied to find all solutions.

4 Evaluation

We have analyzed experimentally the relative performance of the two types of algorithms presented in this paper by using randomly generated CondCSPs. The random CondCSP generator [11] uses d_a, *density of activity*, the probability of generating a non-initial variable as a target variable, and s_a, *satisfiability of activity*, the probability of generating a value in a domain as a condition variable, in addition to the standard parameters of density and satisfiability of the problem compatibility constraints.

The test suites we designed generated different problem topologies with a problem size of number of values per domain $d_{size} = 10$ and the number of variables both fixed at $n = 10$ and varied up to $n = 100$. Density and satisfiability of compatibility and activity were varied. We designed those experiments in order to get problems of a large enough solution space nevertheless of manageable size. The main goal of the experimental evaluation is to analyze the comparative performance of both types of algorithms encoded in the same C++ framework and run on the same machine (1.7GHz Pentium processor).

The rest of activity parameters were preset for all problem instances as follows. The probability that an activity constraint be of inclusion activity constraint was set to 0.5. The maximum condition values per domain $maxCondDom$, total condition values per problem, $totalCond$, and maximum of target variables per condition value, $maxTargetCond$, were computed according to the following formulas: $maxCondDom = s_a{*}d_{size}$, $totalCond = s_a{*}d_{size}{*}n$, and $maxTargetCond = n/2$.

We designed three test suites for our experiments. We compared the execution time, in number of seconds, for $CondFc$, $CondMac$, $GtFc$ and $GtMac$. In the first test suite (Figure 3, Figure 5), density parameters were kept constant, $d_c = 0.3$ and $d_a = 0.5$, while satisfiability parameters were varied: $s_c \in [0.2 \ldots 0.4]$, and $s_a \in [0.5 \ldots 0.8]$ (in increments of 0.1). In the second test suite (Figure 4), compatibility density was increased to $d_c = 0.5$, and the variation of the compatibility satisfiability was shifted up into the interval $s_c \in [0.4 \ldots 0.7]$. The activity parameters were maintained the same as in the first test suite. In the last test suite, we use one parameter setting combination in the first test suite, $d_c = 0.3$, $s_c = 0.4$, $d_a = 0.5$, $s_a = 0.6$, and vary the number of variables, in the interval $n \in \{10, 20, 30, 40\}$.

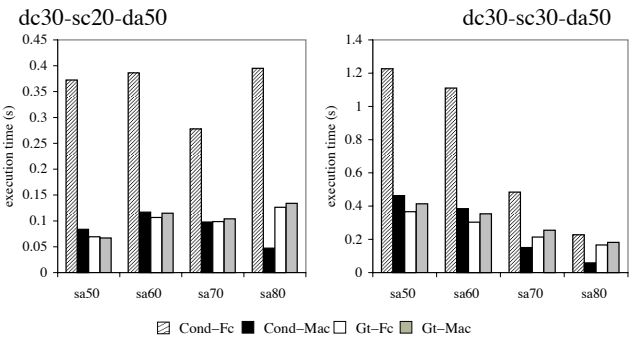


Figure 3. Execution times (seconds) for variable satisfiability, $s_c \in \{0.2, 0.3, 0.4\}$ and s_a in $[0.5 \ldots 0.8]$, and fixed density, $d_c = 0.3$ and $d_a = 0.5$.

From the first test, we considered the experiments with the highest satisfiability of compatibility in order to exhibit additional measurements: number of solutions, number of backtracks, as well as number of condition checks. In the $Cond$ algorithm, number of backtracks measures the standard number of backtracks during enumeration of solutions. In the Gt algorithm, the total number of backtracks is composed of the number of backtracks performed while generating the tree of standard CSPs plus the number of backtracks used while solving those standard CSPs. The number of condition checks for $Cond$ is incremented each time an activity constraint is checked for its condition. In Gt, the number of condition checks is incremented when the condition is reformulated into a compatibility constraint.

We observe the following.

1. All test suites show that time performance of all methods improves with increasing satisfiability of activity.
2. In all cases $CondMac$ outperforms $CondFc$, which confirms what has been reported already [8].
3. $GtFc$ and $GtMac$ relative time performance depends on the size of the solution space. In general, underconstrained problems are solved faster with $GtFc$. The first two suites (Figure 3, Figure 4) show that $GtFc$ becomes faster than $CondMac$ when the satisfiability of compatibility is increased, and, within the same parameter setting problem class (second graph in Figure 4), $GtFc$ runs faster with higher satisfiability of activity.
4. In terms of algorithm effort measured by the number of backtracks, $GtFc$ and $GtMac$ exhibit the same effort. The number of condition checks in the Gt algorithm is considerably smaller than the one for the $Cond$ algorithm.
5. For overconstrained problems ($n = 30$, $n = 40$, as shown in Figure 6), performance improves with increasing number of variables. Both $CondMac$ and $GtMac$ perform better than their Fc counterparts.

The following factors contribute to support these observations.

1. The number of solutions decreases with increasing satisfiability of

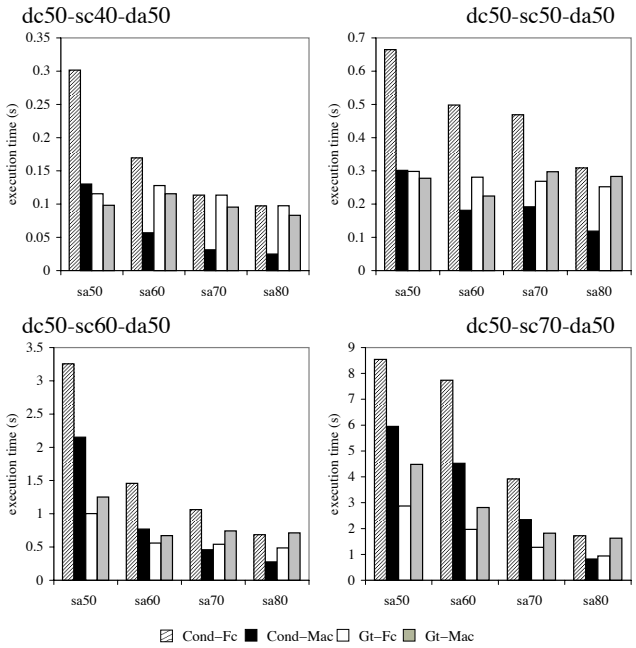


Figure 4. Execution times (seconds) for higher $d_c = 0.5$ and range of $s_c \in \{0.4, 0.5, 0.6, 0.7\}$ (activity parameters unchanged).

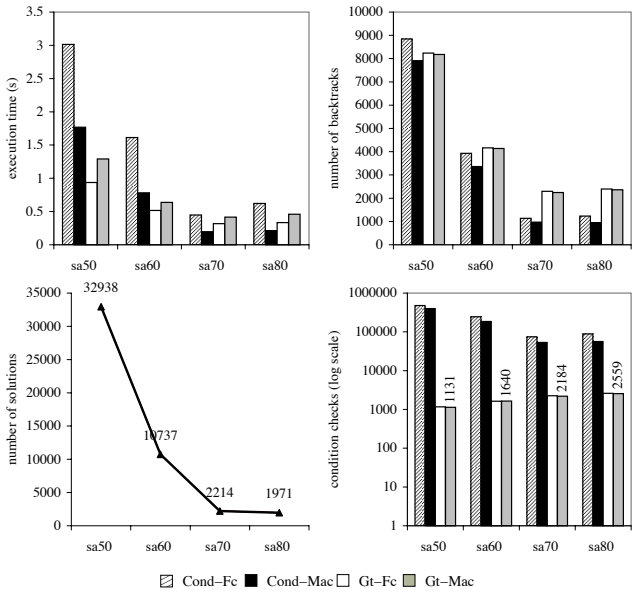


Figure 5. Execution times (seconds), number of solutions, and effort counts (number of backtracks and condition checks) for $d_c = 0.3, s_c = 0.4, d_a = 0.5$ and s_a in $[0.5 \ldots 0.8]$ range.

activity, which makes problems easier to solve when algorithms searched for all solutions.

2. $CondMac$ takes advantage of activity constraints and the tension between inclusion and exclusion activations to prune variable domains. Since the experimental studies use binary CondCSPs, the activity constraints with unary conditions yield a good pruning effect.
3. Gt transforms all exclusion constraints into compatibility constraints prior to the tree generation. Thus, the advantage of $MacActivity$ pruning in detecting activation conflicts does not manifest in Gt. Secondly, it is known that Mac's performance deteriorates with increasing problem density. In the case of Gt, the compatibility density of the resulting standard CSPs is larger than the original CondCSP's compatibility density, since all activity constraints have been reformulated into compatibility constraints. Overall, the pruning power of $MacCompatibility$ is overcome by the implementation overhead which, in the end, does not always pay off. Consequently, $GtFc$ does more efficient pruning at a lower effort cost.
4. Execution times and effort shown in the number of backtracks and condition checks decrease with increasing satisfiability of activity.
5. The better pruning of maintaining arc consistency for both $Cond$ and Gt pays off while solving overconstrained problems.

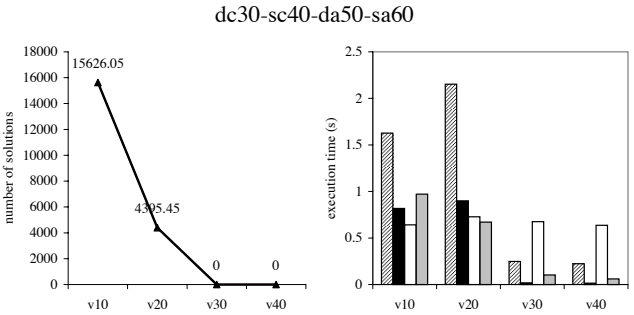


Figure 6. Execution times (seconds) for variable number of variables, $vars \in \{10, 20, 30, 40\}$ for problem populations with $d_c = 0.3, s_c = 0.4, d_a = 0.5, s_a = 0.6$.

5 Conclusion

In this paper, we have introduced a CondCSP solver that includes two algorithms for solving CondCSP. We have used random CondCSPs and designed test suites for varying problem topologies. Our evaluation analysis shows that there is not one winner, but that reformulation solving, Gt, in conjunction with forward checking performs better on large problems with larger solution spaces in the tens of thousands than the direct solving methods. Direct solving with maintaining arc consistency is more efficient on larger, overconstrained problems. Therefore, we suggest that a combination of both algorithms could improve efficiency further. In addition, a more systematic study with an extensive coverage of problem topologies should be conducted in the future.

REFERENCES

[1] Minh Binh Do and Subbarao Kambhampati, 'Solving planning-graph by compiling it into CSP', in *Artificial Intelligence Planning Systems*, pp. 82–91, (2000).
[2] E. Gelle and B. Faltings, 'Solving mixed and conditional constraint satisfaction problems', in *Constraints, 8(2):107–141, 2003.*, (2003).
[3] E. Gelle and M. Sabin, 'Solving methods for conditional constraint satisfaction', in *In Papers from the 2003 IJCAI Workshop on Configuration*, (2003).
[4] Esther Gelle, *On the generation of locally consistent solution spaces*, Ph.D. Thesis, Ecole Polytechnique Fédérale de Lausanne, Switzerland, 1998.
[5] Felix Geller and Michael Veksler, 'Assumption-based pruning in conditional csp', in *Principles and Practice of Constraint Programming*, (2005).
[6] I. Miguel, P. Jarvis, and Q. Shen, 'Flexible graphplan', in *Proceedings of the Fourteenth European Conference on Artificial Intelligence*, pp. 506–510, (2000).
[7] S. Mittal and B. Falkenhainer, 'Dynamic constraint satisfaction problems', in *Proceedings of the 8th National Conference on Artificial Intelligence*, pp. 25–32. The MIT Press, (1990).
[8] M. Sabin, *Towards Improving Solving of Conditional Constraint Satisfaction Problems*, Ph.D. dissertation, University of New Hampshire, Durham, NH, U.S.A., 2003.

[9] M. Sabin and E.C. Freuder, 'Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems', in *CP'98 Workshop on Constraint Problem Reformulation*, Pisa, Italy, (October 1998).
[10] Mihaela Sabin, Eugene C. Freuder, and Richard J. Wallace, 'Greater efficiency for conditional constraint satisfaction', in *Principles and Practice of Constraint Programming*, pp. 649–663, (2003).
[11] R.J. Wallace, *Random CSP Generator*, Constraint Computation Center, University of New Hampshire, Durham, NH, U.S.A., 1996.

Configuring from Observed Parts

Lothar Hotz [1]

Abstract. This paper presents a conceptual framework that allows the configuration of aggregates using existing parts observed in reality. Typical problems that can be solved with such an approach are recognition problems that construct aggregates from given observations. One instance of the conceptual framework is given by the system SCENIC, which allows the interpretation of video scenarios showing table-laying scenes.

1 Introduction

Configuration is the construction of aggregates using parts of a certain domain. In a typical configuration schema, starting from a given goal object that represents the aggregate to be configured, this aggregate is decomposed in its parts step by step. Thus, a goal-driven, top-down approach is typically used for constructing a configuration. Furthermore, the parts that should be in the aggregate are previously described in a configuration model. This configuration model describes implicitly all configurations that can be constructed. Typically a knowledge acquisition process creates that configuration model. A knowledge engineer acquires existing components to be configured and formalizes them in the configuration model. In a second step, the configuration process uses the configuration model for configuring new products. During this configuration process the configuration model is seen as fixed. If parts exist, that should be integrated into the aggregate, they are defined in a task specification on the basis of the configuration model before the configuration process starts. No further existing parts are considered, beside those. Thus, during the configuration process the real existing parts are only considered indirectly through the configuration model. The configuration process relays on the fact that the real world is in the state as it was during knowledge acquisition. If new parts are created in the real world during the configuration process, those components are not taken into account. Only if the configuration model is updated (e.g. by a third step - an evolution step), those new parts could be taken into account. Summarizing, we speak of a three-step approach — knowledge acquisition, configuration process, evolution process. The relationship between the configuration model and the real world is only established during knowledge acquisition, the task specification and probably an evolution process, but not during the configuration process. An example of such an approach is given in [6].

In the standard configuration schema, as often in knowledge-based systems, a gap exists between the real world and the configuration model. However, this schema is suitable for non dynamic, technical domains, e.g. hardware configuration [8, 2, 7, 11, 9]. For situations with a number of changes (e.g. software configuration) or even dynamic situations (e.g. interpretation of video scenes) this three-step approach can be improved by taking into account the parts that can be observed during the configuration process.

In this paper, we discuss the *configure-from-observed-parts* problem, or *CofOP-Problem* for short. This problem focuses on configurations that can be derived from real existing and observed parts at a first place. Only in a second step requirements are included to compute the desired configurations. In this case, it can be ensured that the resulting configuration has really counterparts in the real world.

In Section 2, we have a closer look at the CofEP-Problem. Section 3 focuses on structural configuration as a starting point. As the core of the paper, we present a conceptual framework (see Section 4), a general system (see Section 5) as well as a concrete instance of such a system, a system for interpreting video scenes (Section 6). The paper concludes with a discussion (Section 7) and an outlook in Section 8.

2 Problem Description

The *configure-from-observed-parts* (CofOP-Problem) can simply be rephrased with following questions:

1. Given a number of parts, what aggregates can be built with these parts? What can we do with the things that exist?
2. Now, we have a partial aggregate made from existing parts, what is missing to make this aggregate to fulfill given requirements?
3. What role do aggregates play in finding a final configuration? What customer requirements can be fulfilled with the aggregates that are constructed?

In the CofOP-Problem we do not start the configuration from requirements and than try to infer needed parts, but we start from the things that exist and infer the use of the constructed aggregates.

Typical application examples are those where it is not clear in the beginning what should be configured, or what the aggregate will look like. In these cases, goal objects can not be given, but only the parts, which should form the aggregate to be constructed. Thus, a more explorative task has to be performed for handling the CofOP-Problem. Examples are:

- Systems that interpret images; existing parts in this case are shapes or objects an image processing system can identify in the images.
- Systems that interpret videos; existing parts in this case might be tracked objects that can be identified by a tracker (e.g. SCENIC see Section 6).
- Systems that build software; existing parts in this case are software components provided by an asset store or software configuration management system.

An extension of the CofOP-Problem is given with the question:

4. What happens if a new part is observed during the configuration process?

[1] HITeC e.V., University of Hamburg, Germany, email: hotz@informatik.uni-hamburg.de

In this case, not only previously given parts are considered in the configuration but parts that are dynamically created. Examples are parts that are created in a development process or dynamic situations of a video film, which should be interpreted.

3 A Structural Configuration System

A typical structural configuration system designed to support the configuration of aggregates based on component descriptions in a knowledge-base is organized in four separate modules:

Concept Hierarchy *Concepts* are described using a highly expressive concept description language, and embedded in a taxonomical hierarchy (based on the `is-a` relation) and a compositional hierarchy (based on the structural relation `has-parts`). Parameters specify concept properties with value ranges or sets of values. *Instances* selected for a concrete configuration are instantiations of these concepts.

Constraints *Constraints* pertaining to properties of more than one object are administered by a constraint net. Conceptual constraints are formulated as part of the conceptual knowledge base and instantiated as the corresponding objects are instantiated. Constraints are multi-directional, i.e. propagated regardless of the order in which constraint variables are instantiated. At any given time, the remaining possible values of a constraint variable are given as ranges or value sets.

Task Description A configuration task is specified in terms of an aggregate which must be configured (the goal) and possibly additional restrictions such as choices of parts, prescribed properties, etc. Typically, the goal is the root node of the compositional hierarchy.

Procedural Knowledge Configuration strategies can be specified in a declarative manner. For example, it is possible to prescribe phases of bottom-up or top-down processing conditioned on certain features of the evolving configuration.

A stepwise configuration could be executed by the configuration system according to the following basic algorithm:

```
Task specification
Repeat
   Determine current strategy
   Determine possible configuration steps
   Select from agenda and execute one of
          { aggregate instantiation,
            aggregate expansion,
            instance specialization,
            parameterization,
            instance merging }
   Propagate constraints
   Check for conflict
```

An aggregate is completely configured if all properties of the aggregate have been parameterized, all its required parts have been completely configured, and all constraints are satisfied. A conflict is encountered when the constraint net cannot be satisfied with the current partial configuration. In this case, automatic backtracking occurs. Backtracking can be controlled by procedural knowledge to achieve "intelligent backtracking". An example of such a system is KONWERK [1, 3]. For a more general ontology for configuration see [10]; for an overview of standard approaches see [4, 13].

4 Conceptual Framework

The CofOP-Problem requires a distinction of the following aspects[2]:

The reality (*Realität*) are things in the outside, materialized world. In our case, the real existing parts that can be used for configuring span the reality.

Substantiality (*Wirklichkeit*) covers everything the system knows about the reality. From the view point of the system the substantiality constitutes the real world. The substantiality is divided into the outer and inner substantiality.

Evidence space (the outer substantiality, perceived reality) (*Äussere Wirklichkeit*) is the representation of the reality in the system. The real world is *reflected*. The evidence space represents the observed parts that can be used in the configuration. Objects in this space are called *evidence*. Evidence is seen as given and not alterable.

Hallucination space (the inner substantiality) (*Innere Wirklichkeit*) is the representation of the things that might exist, i.e. that can be imagined. Objects in this space are called *real objects* because seen from the system point of view, those objects constitute the real world, even if they do not exist. Each real object might have or might not have evidence; it might or might not be justified in the evidence space, i.e. in the reality. Thus, real objects can be justified by evidence or can be imagined (*hypothesized* or *hallucinated*). Real objects that are imagined (i.e. have no evidence) are called *imagined objects* or *hypotheses*. Those real objects that are in fact in the reality (i.e. have evidence) are called *perceived objects*. Imagined and perceived objects can be distinguished from each other. However, real objects, if justified or not, can be part of aggregates, thus, can form the resulting configuration. In the hallucination space new inferences about the substantiality can be made.

The domain-dependent configuration model (*die Bedeutung*) contains the knowledge about a certain domain, typically represented in a knowledge base, i.e. configuration model. This model is used to interpret the substantiality and to infer new real objects.

5 Technical Aspects

In this section, technical issues concerning the realization of a system that solves the CofOP-Problem are presented.

5.1 Architecture

The general architecture consists of two systems, a low-level sensoric system and a high-level inference system. The low-level system establishes the connection to the reality. Examples are:

- an image processing systems which computes shapes or objects of an image;
- a warehousing system providing a constantly updated, complete description of all existing parts of a technical system;
- a software asset store or configuration management system providing existing software components.

The output of the low-level system is mapped to the evidence space and represented as evidence in the high-level system (see Figure 1).

The high-level inference system is a configuration system, that employs a certain upper model. This upper model represents the conceptual framework described in Section 4. The upper model consists of:

[2] This distinction is inspired by Spitzing's distinctions made in the photography research, see [12].

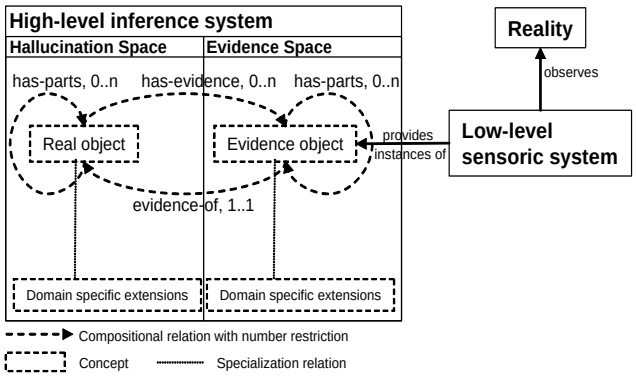


Figure 1. General architecture including the upper model representing the conceptual framework.

- the concept `evidence-object`. Its instances represent the parts that exist in the reality, i.e. its instances form the outer substantiality. Evidence-objects can have parts, if the low-level system is able to observe those, or have no parts, if the low-level system can only observe non-structured objects, which is typically the case.
- the concept `real-object`. Its instances and parts represent all what is known and what is hallucinated by the system, i.e. its instances form the inner substantiality.

The concepts `evidence-object` and `real-object` are related to each other by the relation `evidence-of` and its inverse `has-evidence`. A real object may have evidence or may have not, an evidence object should always be related to a real object; both relational aspect are represented with number restrictions. Real objects with a `has-evidence` relation of cardinality 0 represent imagined objects, real objects with a `has-evidence` relation of cardinality greater 1 represent perceived objects. Thus, there are no concepts for imagined or perceived objects, because every real object can be in both roles for distinct situations.

The relation `has-evidence` has following semantics: If an instance of the concept `evidence-object` exists, than an appropriate instance of the concept `real-object` is created. However, if an instance of the concept `real-object` exists *no* instance of the concept `evidence-object` is created, because evidence objects represent the reality, i.e. only the low-level sensoric system can trigger the creation of such instances. Thus, they represent imagined objects as long as no evidence objects are observed by the low-level system.

Domain dependent configuration models are added to the upper model by specializing the concepts `real-object` and `evidence-object`. The specializations of `evidence-object` strongly depend on the low-level sensoric system's facilities and the data it can deliver, e.g. structured objects or no structured objects, types of objects or descriptions via properties.

The relation `evidence-of` establishes a mapping from the evidence space into the hallucination space. This mapping is domain dependent. The mapping has to be made with the information provided by the evidence object.

5.2 Operational Aspects

The configuration steps introduced in standard configuration systems can be used for configuring an aggregate with observed parts (see Section 3 and [5]). However, following aspects are of major importants for handling the CofOP-Problem:

Configuration from bottom to top, from parts to aggregates: Processing integration steps from parts to aggregates is the first step performed when using observed parts. For example the configuration of the `evidence-of` relation for an evidence object is an integration step.

The bottom-up steps can continue into the hallucination space as long as unique decisions can be made. Thus, all inferences that are justified by the reality are drawn. One may think of this phase as the determination of those things that can be inferred from the evidences; or the answer to the question "What can be done with the parts that exist?" (i.e. Question 1. of Section 2).

Top down for generating hypotheses: In this second step a goal-object can be selected, which represents certain requirements. Starting from this goal-object the normal configuration process is performed. For example, a specialization is done by selecting an appropriate sub concept, which represents a hypothesis made from top-down. Through decomposition further hypotheses are generated and integrated in the configuration. One may think of this phase as an exploration of high-level concepts, which might be responsible for the objects and occurrences observed so far. Furthermore, this phase identifies such parts that are missing for constructing a complete configuration. Missing parts are those real objects that do not have evidence, i.e. the imagined objects (Question 2. of Section 2).

Use of bottom-up generated objects: If through top-down analysis a required part is identified, it has to be checked, if an appropriate real object already exists in the configuration, because it could have been generated in the bottom-up phase. This operation creates a configuration that contains both, imagined real-objects and evidence-based, perceived real-objects (Question 3. of Section 2).

Fuse real objects: If a part is generated from top-down and one from bottom-up, it could be the case, that those two parts are in fact the same. In this case only one part should be in the resulting configuration. If these generated parts fulfill a certain fuse condition, which ensures that they should really be one object, these parts can be fused. This means, that one real object is created, which contains all relations and parameters of the former two parts. The parameter and relation values of the fused object are computed by building intersections. This means that real objects that are related to the former two parts are also considered to be fused (*deep fuse*).

Spontaneous instantiation: To handle dynamic aspects a configuration step is needed that instantiates an arbitrary concept of the evidence space. For example, if a new objects enters a video scene, a new evidence instance should be created. Thus, the low-level system and the high-level system should be synchronized by synchronization points or steady polling. When new evidence instances are created at time t, those have to be integrated in the configuration even if the defined configuration procedure would focus on another configuration step at time t. Furthermore, backtracking has to be initiated, because the inferred conclusions could be wrong, because the evidence base is changed (Question 4. of Section 2).

Backtracking to the hypotheses generation: During configuration conflicts might occur as seen above. Because in the described schema the bottom-up steps are seen as unique inferences from the evidences, only the top-down hypotheses generation can be backtracked. The first hypothesis which was made can be seen as a backtracking point. While the evidence instances including their inferences stay in the configuration, all made hypotheses are withdrawn and the configuration process proceeds with another hypothesis.

6 SCENIC - an Example Applications

In this section, we shortly present the experimental system SCENIC (SCENe Interpretation as Configuration) which utilizes configuration technology for concrete scene interpretation experiments [5].

The SCENIC System was built employing the conceptual framework presented in this paper. The example scenes are taken from a table-laying scenario: A video camera is installed above a table and observes a tabletop. Human agents, sometimes acting in parallel, place dishes and other objects onto the table, for example, cover a dinner-for-two. It is the task of the scene interpretation system to generate high-level interpretations such as ”place-cover” or ”lay-dinner-table-for-two”. Occurrences of this kind are complex enough to involve several interesting aspects of high-level scene interpretation such as temporally and spatially constrained multiple-object motion, a knowledge base with compositional structure, and the need for mixed bottom-up and top-down interpretation steps.

According to the conceptual framework we introduced above, SCENIC consists of a low-level system, in this case an image processing system. Via a video camera and a tracking system the low-level system observes the table. A so called *geometric scene description* (GSD) is generated by the low-level system, which represents the objects and their properties seen in the video film. The high-level system consists of KONWERK and a configuration model containing the presented upper model and a domain specific model for table-laying scenarios (see Figure 2). KONWERK performs the symbolic interpretation subtask in SCENIC.

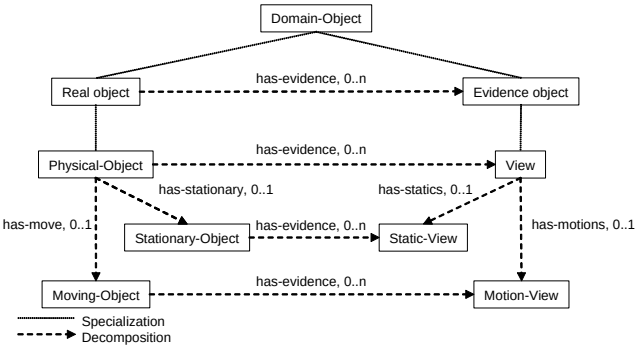


Figure 2. An extract of the domain specific specialization of the upper model presented in Figure 1.

From the GSD evidence objects are created during configuration. The GSD contains the observed objects as well as their motion in time and space. The bottom-up strategy integrates the evidence objects into real objects and those into aggregates of the hallucination space. For example, in SCENIC transports (like a hand-plate transport) are identified during this phase. When the bottom-up strategy is exhausted, top-down expansion begins. In SCENIC a hypothesis about a possible table-scene is generated as an aggregate, e.g. `dinner-for-two`. This aggregate is decomposed and further hypotheses are generated. This way hypotheses about a further development of a scene are generated by the typical configuration steps (e.g. aggregate decomposition see Section 3). During this expansion of possible occurrences the real objects created by the bottom-up strategy are used if possible. Thus, a configuration or *scene interpretation* is constructed, which consists of real objects with evidence and those without evidence, i.e. imagined and perceived real objects. In Figure 3 perceived objects are in realistic shape, imagined objects with possible positions are shown with icons.

A fuse operation in SCENIC occurs, when the low-level system cannot provide distinct information about an object. For example, a certain object can only be identified as a *dish*, not as a *cup* (e.g. the cup on the right in Figure 3 cannot clearly separated from the saucer underneath). From top-down it is inferred that this object has to be a cup, because the hypotheses generation determines that at this certain position of a cover only a cup can exist. At this point a real-object

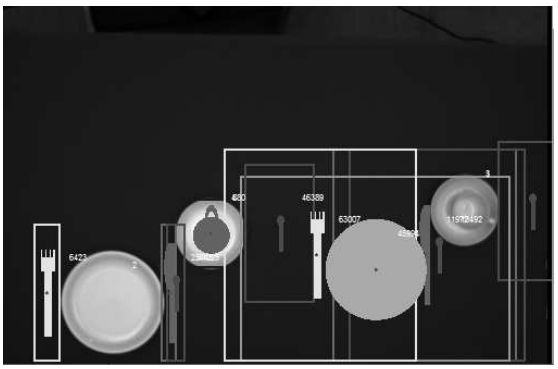


Figure 3. Result of a scene interpretation: realistic shape for evidence objects, hypothesized objects including their possible spatial regions are represented with icons and rectangles.

is created from top-down, representing a hypothesized cup, and a real-object is created from bottom-up, representing a dish. Those two real objects can be fused, if a fuse condition (like position and time overlap) is satisfied. This fusion represents the insight of the system, that the previously not identifiable dish is in fact a cup, given certain hypotheses. This possibility to disambiguate low-level sensoric input through high-level inferences is unique to this approach.

SCENIC was used to interpret video scenes consisting of about 300 frames. During this sequence 28 evidence objects (including those evidence objects representing motions) are processed, 81 imagined real-objects are created for the hypothesis *dinner for two* (again including those representing motions). For keeping track of spatial and time relations 130 constraints are created during configuration. 51 interpretation steps were needed to obtain the first intermediate scene interpretation (see Figure 3), using 90 sec of CPU time (1.8 GHz PC). Backtracking and additional 8 interpretation steps were needed to arrive at an alternative intermediate interpretation for the hypothesis *single dinner* using additional 45 sec of CPU time.

The configuration model consists of 50 types of real objects and 34 conceptual space and time constraints. The operational aspects presented in Section 5.2 are covered by 8 strategies.

7 Discussion

Data-driven approaches may take existing parts into account, if they refer to them during the configuration process. Data-driven approaches perform bottom-up construction of aggregates from their parts. If those parts are identified during the configuration process, a similar scenario as described in this paper has to be handled. However, the conceptual framework presented in this paper clearly separates hypotheses and evidences. Thus, following four situations can be distinguished:

1. Hypothesis, which is supported by an evidence, i.e. the relation `has-evidence` is established (*perceived object*).
2. Hypothesis, for which no evidence is yet found. This is due, when the real object has no relation to an evidence object (*imagined object*).
3. Evidence, which is already interpreted, i.e. the relation `evidence-of` is established (*accepted evidence*).
4. Evidence, for which it is not yet decided how the evidence should be interpreted. This is due, when the evidence has no relation to an imagined object (*not accepted evidence*).

For data-driven approaches employing a schema, where evidences are instances of concepts, case 2 cannot be represented and be used

for reasoning, because no distinguishable instances for evidence objects and hypotheses exist.

A further opportunity to use the presented conceptual framework is its ability to support the evolution process, e.g. in software configuration. To handle the question: What happens if a new observed part comes into play during the configuration process that has no mapping into the hallucination space? This would indicate evidence for something which is still unknown for the system (not accepted evidence). However, because the evidence-object is generic each new developed part (e.g. a software component) can be represented by an instance of `evidence-object` and thus can be considered *during* the configuration process. If such a part cannot be mapped to a real object the configuration system would raise a conflict. Thus, this conflict could be handled by evolving the configuration model according to the new developed part [6].

8 Summary and Outlook

The configuration from observed parts is a problem in the area of recognition or interpretation of images and videos. But also in technical configuration like elevators or software-intensive systems a representation of observed parts, which should be used for configuration, is of importance. In this paper, we present a conceptual framework that explicitly represents observed parts and distinguishes them from other parts, without an existing representative in the reality.

An example for an instance of the conceptual framework is given by the system SCENIC, which interprets video films about table-laying scenes. In an upcoming system, which will be used for interpreting images of buildings, photographed from the front or from a satellite, we will also apply the described framework. The goal here is to identify regions in buildings like windows, doors, roofs, chimneys as well as aggregates like row of houses or roofs. Similar to SCENIC from a low-level system geometric scene descriptions are provided, which are interpreted with the high-level configuration system. The imagined real objects will be used to give feedback to the low-level system; for example, to focus on a certain part of the image for identifying a certain detail of a building.

ACKNOWLEDGEMENTS

This research has been supported by the European Community under the grant IST 027113, eTRIMS - eTraining for Interpreting Images of Man-Made Scenes.

REFERENCES

[1] A. Günter, *Wissensbasiertes Konfigurieren*, Infix, St. Augustin, 1995.
[2] A. Günter and R. Cunis, 'Flexible Control in Expert Systems for Construction Tasks', *Journal Applied Intelligence*, **2(4)**, 369–385, (1992).
[3] A. Günter and L. Hotz, 'KONWERK - A Domain Independent Configuration Tool', *Configuration Papers from the AAAI Workshop*, 10–19, (July 19 1999).
[4] A. Günter and C. Kühn, 'Knowledge-based Configuration - Survey and Future Directions', in *XPS-99: Knowledge Based Systems, Proceedings 5th Biannual German Conference on Knowledge Based Systems*, ed., F. Puppe, Springer Lecture Notes in Artificial Intelligence 1570, Würzburg, (March 3-5 1999).
[5] L. Hotz and B. Neumann, 'SCENIC Interpretation as a Configuration Task', Technical Report B-262-05, Fachbereich Informatik, University of Hamburg, (March 2005).
[6] L. Hotz, K. Wolter, T. Krebs, S. Deelstra, M. Sinnema, J. Nijhuis, and J. MacGregor, *Configuration of Industrial Product Families - The ConIPF Methodology*, Aka Verlag, Berlin, 2005.
[7] S. Marcus, J. Stout, and J. McDermott, 'VT: An Expert Elevator Designer that uses Knowledge-based Backtracking', *AI Magazine*, 95–112, (1988).
[8] J. McDermott, 'R1: A Rule-based Configurer of Computer Systems', *Artificial Intelligence Journal*, **19**, 39–88, (1982).
[9] K.C. Ranze, T. Scholz, T. Wagner, A. Günter, O. Herzog, O. Hollmann, C. Schlieder, and V. Arlt, 'A Structure-based Configuration Tool: Drive Solution Designer DSD', *14. Conf. Innovative Applications of AI*, (2002).
[10] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen, 'Towards a General Ontology of Configuration', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (1998), 12*, 357–372, (1998).
[11] E. Soloway and al., 'Assessing the Maintainabiliy of XCON-in-RIME: Coping with the Problem of very large Rule-bases', in *Proc. of AAAI-87*, pp. 824–829, Seattle, Washington, USA, (July 13-17 1987).
[12] G. Spitzing, *Foto Psychologie*, Beltz, Weinheim, Basel, 1985.
[13] M. Stumptner, 'An Overview of Knowledge-based Configuration', *AI Communications*, **10(2)**, 111–126, (1997).

Configuration of Contract Based Services

Juha Tiihonen[1], Mikko Heiskala[2], Kaija-Stiina Paloheimo[2], and Andreas Anderson[1]

Abstract. Satisfying needs of individual customers by mass-customizing services has been proposed. Although configuration, i.e. specifying a product individual as a combination of pre-defined components, is an important way of achieving mass-customization to industrial goods producers, there is relatively little literature on the applicability of the configuration paradigm to services. In this paper we take a step towards understanding if services could be managed as configurable products. The ideas presented in this paper originate both from existing literature and from our co-operation with four companies that participate in our 3-year research project on configurable services and IT support for service configuration. We show that at least in some contract-based service industries configurable services exist and are used for doing business, and we characterize the services, related processes and special requirements on configurators.

1 INTRODUCTION

Configurable products are one way to achieve the benefits of mass-customization. The design of a configurable product specifies a set of pre-designed elements and rules on how these can be combined in a routine manner without creative design into valid product individuals that meet the requirements of particular customers [1,2].

Services are products with a significant service dimension e.g. [3,4]. Research on *configurable services*, and development of *configurators* [5] particularly suitable for these, is relatively limited [6, 7,8,9,10,11,12,13,14,15,16]. It is not known if the special characteristics often attributed to services i.e. intangibility, perishability, simultaneity of production and consumption, and heterogeneity [17] hinder the development and deployment of configurable services.

1.1 Practical Motivation

Services are often adapted according to properties of the customer, other stakeholders or related equipment. “One size does not fit all”. On the other hand, it is not realistic to fully customize for all customers. For example, fully customized insurance terms for each customer would call for uneconomical resources of insurance mathematicians, lawyers, etc. Similarly, high-volume telecommunications services such as business-to-consumer (B2C) mobile and broadband subscriptions cannot be individually modified for each customer as they must be deliverable through automatic platforms. Therefore a mass-customization approach is often desirable.

Companies today outsource ever more diverse functions but don’t want to spend time and effort in the process. Often full customization is optimal from the customer needs satisfaction point of view. But even customers may perceive fully customized solutions expensive and sub optimally accessible, potentially inconsistent and poorly documented. Further, the time and effort sacrifice in specification may be too much. For these reasons, mass customization may be a lucrative option for customers.

A theoretically promising means to achieve the benefits of mass-customization, service configuration, also has practical relevance and potentially wide applications in a number of industrial contexts. Contract based services are an interesting area to study this phenomenon, as opportunity to elicit customer requirements [18] and observe customer behavior may be better in a contractual customer relationship than in a one-off transaction.

1.2 Goals, Research Questions and Method

Many configurator vendors claim support for configuring services [19], yet few examples of configurable services have been documented in scientific literature. Our long-term goal is to allow cost-effective, semi-automatic or even automatic mass-customization and individualization of services through the web by modeling and managing them as configurable service product families. In this study our research questions are:

(1) Can services be modeled and managed as configurable products?
(2) What can be varied in configurable services?
(3) What processes are related to configurable services?
(4) Do configurable services pose any special requirements on configurators?

In this work we concentrate on services that are performed on the basis of a contract. We considered such services to potentially benefit most of the application of the configuration paradigm and configurator support.

We used qualitative case studies as our method, the four case companies are service providers participating in our 3-year research project on configurable services and their IT support. We have conducted empirical studies through participant observation and open semi-structured interviews in the companies. Further, we experimented modeling some of their service offerings using a configurator designed for physical goods.

Two cases are services offered by manufacturers of configurable equipment: industrial process equipment maintenance services, and information services of configurable heavy industrial equipment, both in business-to-business (B2B) setting. The third case involves insurance and other financial services, and the fourth case telecommunications services, both representing B2C offerings. The cases have some special service characteristics: The equipment related cases involve a significant goods dimension. The financial service company has a near maximal service dimension in their products. The telecommunications case has automated service delivery.

[1] Helsinki University of Technology, Software Business and Engineering Institute, P.O.B. 9210, FI-02015 HUT
[2] Helsinki University of Technology, Laboratory of Work Psychology and Leadership, P.O.B. 5500, FI-02015 HUT
[1&2] firstname.lastname@tkk.fi

2 CONFIGURABLE SERVICES

In this section we first discuss the general structure of our case services, relate that to research question (1), and proceed to consider what is varied to answer the research question (2).

2.1 General Structure of Configurable Service Products

In the following we describe how the services of our cases can be conceptualized - there is a strong analogy to configuration of physical products, and we studied if the same ideas apply.

With physical products the execution of a configuration process produces a specification of a product individual that specifies a number of components (individuals), their compositional structure, parameter (attribute) value assignments and possibly connections between the individuals, see e.g. [20].

In contract based services we were able to identify service elements that corresponded to components. Some service elements were parametric (configurable) and compositional structure could be identified. However, the compositional structure was simple and shallow in our case products. In the compositional structure (at configuration model level), *optional service elements* that can be included or left out were common and *alternative service elements* that are mutually exclusive were encountered. For example, the broadband connections have optionally available SMS-sending via Internet, and increased space for e-mail. An example of alternative service elements in broadband subscriptions is security, the customer can select no security or one of 3 alternatives: virus scanner, virus scanner + firewall, or virus scanner, firewall and spam + content filter. An example of a configurable parameter in the maintenance case is a guaranteed response time in case of a breakdown. It can be selected from 2, 4 or 8 hours. In maintenance services, assisting work-force for official inspections arranged by service provider is an optional service element.

The case services formed "service product families" where the individual members were similar but different in some respects. The general compositional structure was almost identical and same parameters applied to (most) members of the family. A specific service element can be always *included*, *available* optionally or as an alternative, or *not available* at all in some products or service elements. Each service product family contains major fixed service elements, typically the core service and some bundled additional service elements. For example, in our case ISP service products for consumers, speed and connection technology determines a product in the offering, e.g. 512kbit/s ADSL is considered one product and 1 Mbit/s / 256 kbit/s Cable broadband is another product. E-mail, a dictionary, an encyclopedia, internet- news, and IRC services are included in all the products. Often more expensive service products include additional bundled service elements available for additional price or not available at all in lower-end service products. For example, a fast broadband connection includes free access to an electronic phonebook that is not available in the slowest (and cheapest) connections. Similarly, applicable parameters and parameter domains can change by product or service element. E.g., the availability of response times depends on the selected service product – the fastest response times are available only with the more comprehensive maintenance contracts, and the minimum availability of maintained equipment is not specified in a basic maintenance contract.

The case products have few requires- or incompatible-relationships between service elements or characteristics. Relationships of products, service elements or characteristics to customer and/or equipment characteristics are common. For example, availability or pricing of some characteristic value may depend on the related customer. We did not encounter any need for resource-constraint-type of modeling. Need for connections or topological modeling was identified only in the sense that allocation of some responsibilities to different stakeholders could be modeled as connections to objects representing appropriate stakeholders. As these stakeholders can be present in several roles, one way to model them only once but in multiple roles could be through connections.

The above characteristics lead as to conclude that the described case service offerings can be considered configurable.

2.2 Variation in Configurable Service Products

Based on our cases, service products can be varied on a broad spectrum of issues within a predefined envelope of variety. Following the characterizations of Dumas et al. [21], we look at variation through the classical W's, including what, when, who, where, how, by whom, and why. A service element or parameter in a service can relate to several of these views. For example, a broadband connection must always be installed. Therefore selecting if a turnkey installation is performed affects both the what-view (the scope of service), and the who-view (who performs the installation).

2.2.1 What-variation

Often what-variation relates to the scope of the service: are some optional elements included or which of alternative scopes is selected. Some examples were given above. Further, insurance policies may vary on what is covered, against what risks, and on maximum coverage.

The what-view may also relate to pricing: what is included in the periodical fee, and what is charged on by-use basis. For example, maintenance contracts have a number of alternative amounts of repair work covered by a periodical fee.

2.2.2 When-variation

When-variation relates to the temporal aspects of a service or some of its elements. Such aspects include availability, pricing or response performance. For example, in the maintenance cases evening or weekend repairs can be selected or left out. The temporal aspects may affect the whole service or some of its elements. For example, in our maintenance case emergency services are available all the time, but regular repairs may have more limited temporal coverage.

When-variation can also relate to response-times. For example, in maintenance services it is possible to specify whether repair begins within 2, 4 or 8 hours after breakdown.

2.2.3 With what? Who? How?

The human and physical resources used for a service and assignment of responsibilities to different stakeholders offer sources for variation. Further the way some service elements are delivered may be varied.

In our broadband case there are two main technologies for core service delivery – ADSL via telephone network and cable modems via cable network. These can be seen as configurable method for service delivery.

In the insurance case there is a budget-oriented car insurance product where repairs are performed with third-party parts and the insurance company selects the repair shop. In normal cases original parts are used and the insurance holder may decide where the repair is performed.

In our cases, by who-variation was related to the scope of service

– the what-view. In other words, some element of the service may be assigned to the service provider or to the customer.

In our cases, reporting and payment are sources of with-what and how -variation. In maintenance services it is possible to specify with what and how stakeholders are informed about major maintenance events. For example, e-mail and/or SMS can be sent to specified stakeholder(s) when a breakdown has been repaired. Billing can be configured to be electronic or regular paper-based, and payment can be regular or direct-debit.

2.2.4 To Whom –variation

The service recipient –be it a human or equipment- is specified always in our cases. A service product may have relations to a number of stakeholders that may or may not be explicitly defined. Actual variation of the service based on the to-whom view is less obvious in our cases. In some cases the delivery process is affected - e.g. security regulations may require two service technicians instead of one to perform some tasks if specific properties are present in the equipment to be maintained. Further, some service elements or possible values of characteristics are targeted to specific segments or types of customers. In addition, the availability of some service elements may depend on properties of the customer and/or equipment. For example, all-inclusive maintenance contracts are not available for old equipment, and medical insurance may not be available to persons above a specific age.

2.2.5 Where –variation

Service delivery location may be a source of variation and have significant effect on total customer sacrifice. Some training services of one of our cases can be configured to take place at customer premises or at service provider's premises. Large equipment is maintained on-site, but for smaller equipment a choice may be offered.

2.2.6 Why –variation

We did not encounter any explicitly why-view related sources of variation in the configurable service offerings.

2.3 Specific common sources of variation

In this subsection we discuss some sources of variation that may be present in many different types of configurable service products. These include pricing models, information and reporting, paying and billing, ownership and intellectual property rights, service quality attributes, and loyal customer benefits.

Pricing models for services and products are a complex phenomenon, a related body of literature has been analyzed e.g. by Miranda [22]. We encountered three basic types of price elements: one-time, recurring (periodic), and pay-by-use. *Initiation price elements* are paid once, typically when the service contract is initiated. For example, telecommunications services often have an initiation fee. *Periodic price elements* such as monthly or yearly fees are common in our cases. *Pay-per-use price* elements are also common. For example, mobile phone calls may be charged by use.

Allocation of total service cost to different kinds of price elements varies significantly. In our case services initiation fees are relatively insignificant. Allocation to recurring and pay-per-use elements varies significantly. In minimal mobile subscriptions without bundled phone calls or other extras the periodic (monthly) price element is small and basically just covers that mobile services are available and billable. At the other extreme, periodic payments in insurance services cover the whole service fees.

In our cases each service product has an associated pricing scheme that can be fixed or configurable. A pricing scheme may contain initiation, periodic and pay-per-use elements. Often different combinations of periodic and pay-per-use are offered –increased periodic payments include increased amount of use or offer reduced pay-per-use rates.

Sometimes a number of configurable service products differ significantly only in pricing. For example, a mobile subscription may have a specific price when calling to the same service operator's network, and a different price when calling to other networks. Another mobile subscription may have a flat rate to all networks. These different service products can be configured to behave exactly the same way except for pricing.

Information and reporting can offer significant value, or when performed poorly, increase significantly total customer sacrifice. Here too, one size does not necessarily fit all. Information and reporting on services are thus a potential source of variation. In our maintenance case, configurable notifications from service events help the customer-side to be informed on the status, e.g. in case of equipment breakdown. The scope of information and reports available to customers via extranet can be configured. Even alarms on repair costs exceeding a pre-determined value or number of faults can be provided.

Paying and billing are also sources of variation. To some customers of our case companies a configurable number of payments and due date may offer extra value. Bills may be standard paper-based or electronic, and payment options may include e.g. direct-debit in addition to regular payments. Information on what forms the payments (e.g. more detailed itemization of per-use charges) may also offer configurable options. Further, some customers value bills where a number of separate billing targets are billed simultaneously and information is grouped as desired.

Ownership and intellectual property rights (IPR) of information or intangible deliverables can be sources of variation. For example, who owns databases gathered in remote monitoring of equipment or detailed maintenance history? Currently these are not configurable options in our cases, but at least in one company they have required case-specific negotiations.

Service quality attributes such as performance, dependability, security and safety can be sources of variation. For example, basic maintenance contracts do not guarantee availability while higher-end contracts include increasingly higher guarantees on availability. In a similar way, some temporal when-aspects such as how fast repair starts after a breakdown can also be considered quality attributes. Broadband connection speed directly affects the performance of the service.

Various loyal customer benefits can be offered. One of our case companies offers a number of mutually excusive benefit programs.

3 PROCESSES

In this section we discuss the processes related to our case services to answer our research question (3). Again, we see a strong analogy to previous findings in configuration of physical goods [1].

3.1 Sales / specification process

Contract based services in our cases have a similar sales phase (specification phase) as configurable goods where the service along with it's price is specified. The configuration task produces a contract and possibly some non-contractual additional information

elements.

Based on our mystery shopper experiences and interviews at our consumer market companies, and to some degree in maintenance contract sales, current sales processes have several challenges.

Sometimes the sales process tends to be product-centric. The persons at customer interface may start introducing and selling individual service products instead of analyzing the actual needs or requirements of the customer. For example, in a number of cases a potential customer who had made an appointment for comprehensive analysis of insurance-related needs was met with a clerk who started selling some specific insurance policy for a specific (assumedly) needed coverage.

Service product options considered less important by the person at customer interface may not be offered at all. For example, mobile subscriptions include a significant amount of optional value-added services of which only a small subset were offered.

Consultative mode of selling is felt desirable in at least two of our case companies. The idea is to find out relevant properties of the customer and other stakeholders, related equipment and environment as well-as needs to be able to recommend a suitable service solution. It was felt that this could alleviate some problems of product centric sales events.

Services for consumers were available through several sales channels while B2B maintenance and information services were sold only directly by the service provider. Service pricing to customers had little room for bargaining while the maintenance services were typically priced case by case. In telecommunications services and insurance services a contract proceeded automatically via IT support to delivery process (telecommunications service provisioning, insurance contract activation).

3.2 Reconfiguration

Managing reconfiguration seems to be more important in contract-based services than with most industrial goods. Long-term relationships between the supplier and customer are a norm. Often the service must be adjusted when customer needs, equipment, environment or other relevant aspects change.

On the other hand, management of reconfiguration may be easier in services than in goods, because the primary target of configuration is not a physical product individual. Therefore errors or inaccuracies in as-maintained configuration description, and condition of components are not as relevant. Optimization for maximally using old components is not necessary. Systematic "genuine" reconfiguration instead of project-based modernization requiring design may be possible more often. After a company changes its offering, it may be possible just to mass-update (map) old configurations to corresponding new ones in a way that makes reconfiguration within the new offering possible. Of course, this is not always the case, e.g. if the old configuration is not available in the new offering.

For example, in insurance and telecommunications services systematic reconfiguration is common. Telecommunications customers subscribe to new additional services or change their subscription type. Insurance related needs change and insurance policies need to be updated to reflect these changes.

3.3 Service delivery process

The service elements covered by a service contract may take place once, in discrete service events or continuously. In our cases service delivery process based on a single contract usually has repetitive discrete service events "moments of truth". These discrete service events can occur periodically with fixed periods (e.g. official inspections), periods determined by the customer and/or service provider (preventive maintenance based on a device-specific plan), or on demand (mobile phone calls). Some service elements such as turnkey installation of a broadband connection are performed only once. Insurance coverage or an always "open" broadband connection can be considered as continuous service delivery. Some of our case services include several types of delivery. For example, a broadband subscription user may also use value added services in discrete service events.

As discussed previously, configuration decisions may affect significantly the delivery process: e.g. what is done, when something is done, who manages or decides something, etc. Therefore information flows are important – service delivery process must act based on what was agreed in the specification phase. On the other hand, in one case we identified some parts of service delivery process that are not affected by the service configuration and can be performed without such information.

Service delivery processes of core services in our cases have a very different nature. In telecommunications, configured services are made available (provisioned). After provisioning, service delivery is automatic – the customer can use the service at will using his/her equipment. Even provisioning is (almost) automatic. Routine maintenance is performed for the customer without any active customer participation: the service person often even uses his/her keys and performs the necessary actions without presence or activities of the customer. In insurance services, there is no actual service delivery "if things go well". Instead, there is just a promise to manage financial consequences of specified harmful events.

As exemplified above in the maintenance and insurance cases, contrary to traditional service definitions, customer participation does not always take place. On the other hand, educational services of the other heavy industrial goods manufacturer and turnkey-installation of broadband subscriptions include regular customer participation and simultaneous consumption and production.

Customer participation and role can be directly or indirectly configured – a broadband user may perform the installation or participate in the turnkey installation process.

3.4 Development process

The development process of services is normally separate from individual deliveries in our case companies. When introducing new services, there may be overlap in development and delivery processes. In at least one company the difficulty of piloting new contract based services or features was pointed out due to long-term nature of commitments made.

We have no case experience in these cases on new service development. Therefore there is little we can say on how the companies define the appropriate offering – what variation to offer, and how they develop capabilities to sell, price and deliver them.

4 APPLICABILITY OF CONFIGURATORS TO SERVICES

In this section we consider the applicability of configurators to services to answer our research question (4).

Most of the 30 vendors studied in [19] claim their configurators support services. Only two vendors describe their modeling concepts and neither introduces any service specific concepts. No modeling examples were found.

We experimented modeling of broadband subscriptions [19],

maintenance contracts, mobile subscriptions as well as some insurance policies with WeCoTin Configurator [23] that was designed for configuring goods. Modeling in WeCoTin is based on typed objects (components that can have attributes), compositional structure in a form of generic product structures, and constraints [23].

Modeling of contract-based service offerings as configurable products was possible without significant challenges. Based on our experiments and vendor claims we conclude that at least some configurable service offerings can be modeled and configured with traditional configurators. However, we felt a conceptual mismatch in modeling because thinking in components did not seem natural for services. We did not model prices of the offerings. Instead of one price typical for goods configuration, our telecommunications case would require at least two – the initiation and periodical fees need to be kept separate.

According to our previous modeling experiences of physical products, modeling aspects external to the product itself is not usually needed. However, in our service cases, the customer or other stakeholders and/or related equipment, environment, or their properties must be modeled to verify that some services, service elements or some values for their characteristics are available or that they are priced appropriately. Therefore configuring a suitable service specification can be challenging. Recommendations, warnings, and possibly optimization could be useful. We applied the soft constraint mechanism of WeCoTin to warn when some recommendations are not satisfied. However, we felt that such warnings are displayed too late – we would have liked to have guidance towards the good solutions.

Supporting reconfiguration seems to be a business requirement for configuring some contract-based services. For example, B2C telecommunications services require such support as volumes are so high that automated reconfiguration support is a must. We did not address reconfiguration in our modeling experiments.

In some cases it seems that modeling service delivery processes and resources in the sales phase would be beneficial and would thus require appropriate modeling support.

5 PREVIOUS WORK AND DISCUSSION

In this section we first compare our results to previous work and then briefly discuss some of our findings.

Service configuration based on pre-determined specification options and/or delivery modules, possibly supported with configurators, has been at least a partial goal in several papers. Of these, configuration of maintenance services of industrial goods are discussed in [13,14], configuration of financial services in [10], and insurance in [12,24], and customization of IT services in [15]. Moreover, the ILOG JConfigurator has been used in financial services and insurance configuration [25] and telecommunications services have been configured with the CAWICOMS Workbench and WeCoTin [26,19]. These papers support that services can be managed as configurable products in our case company service domains: financial, insurance, maintenance, and telecommunications services. Further, travel is a domain of interest in [27,28]. Combining services from pre-determined modules is suggested for IT consulting services in [11]. Service configuration in general has been discussed in [9,29] and a configurators intended for both physical and service products were described in [30,19]. The work of [9] has been applied in a case bundling energy services with broadband access [31].

Dimensions of service variation have been discussed by several authors. Different types and sources of variation in services and how to manage or limit their consequences in service delivery have been discussed by Harvey et al. [7] and McLaughlin [32] in service management literature. Their focus is on the management and structuring of the service delivery system and process whereas our focus in this work is on defining the dimensions along which configurable services specifications can vary. The non-functional properties of services have been discussed in [21,33]. Some identified properties such as rights to terminate the contract prematurely were not present in our cases - at least not as configurable characteristics.

The requirements for conceptual modeling of configurable services in configurators are discussed in [29, 9]. Both have a high-level process perspective that is new to configuration conceptualizations. Further, capturing the relevant customer characteristics is stressed in [29], a perspective also absent from previous work. In [34] the mainly goods-based product configuration literature has been reviewed for the benefits and challenges related to product configuration and configurators. The paper provides a conceptual analysis of whether the found issues are relevant in service settings.

Our cases did not offer who-variation on personnel attributes that deliver the service (e.g. qualifications or skills), or with-what characteristics of physical elements or equipment used in service delivery (e.g. quality or sophistication of equipment used). This was considered somewhat surprising as such examples are easy to find from other industries.

Why-variation was not present in the service specifications. Selecting a suitable specification could benefit from the why view, e.g. selecting a broadband subscription based on intended use and existing services can benefit from this. However, obtaining and understanding real customer needs can be difficult [35].

6 CONCLUSIONS AND FUTURE WORK

In this paper we discussed the phenomenon of configurable services from the point of view on service and process variability. Configurable services are one way to offer mass-customization of services. Similarly as in physical products, configurable services fill a gap between fully customized services (e.g. consulting projects and other professional services) and mass-services (e.g. electricity and other utilities for consumers or mass-transit).

We looked at some contract-based services where service delivery takes place on an ongoing basis based on a contract specified as a combination of pre-designed elements. The offerings of some of our case companies can be clearly considered as configurable.

Similar processes as in physical products can be identified in contract-based services. The development process is separate from sales (specification) and delivery based on such a specification. There is a separate specification phase where sales configuration takes place. Service delivery takes place repetitively based on a specification. Reconfiguration seems more significant but often easier than with most physical products.

Configuration modeling based on compositional structure, taxonomy, attributes and constraints can be used for modeling the service offering of our cases. Our case services are easy to configure in the sense that there are few strict constraints on the service itself. However, there are many constrains on what services, service elements, or service characteristics are available or are suitable for customers and/or owned equipment (or their characteristics). Generic product structures can be used in modeling of services. However, the conceptual match to service elements is not direct.

IT supported consultative selling could potentially offer significant benefits to some of our case companies. Adequate IT support for that must be able to deal with strict constraints, recommendations, and possibly optimization.

There are many fundamental subjects requiring further work in

service settings. When service is configuration a viable business option? How to decide what variation to offer? Some of the metrics presented in [35] could provide answers. However, the metrics are geared towards goods and manufacturing. It should be studied if customer participation in production, lack of inventories, intangibility and other service characteristics influence the metrics. How to develop configurable services? How to modularize them? How common and severe are service configuration errors in practice? How configurable services affect the way companies should organize themselves? How do intangibility, perishability, simultaneity of production and consumption, and heterogeneity of services [17] hinder ability develop (and deliver) configurable services? For example, can the experience or personal interaction be configured due to potential variation caused by heterogeneity caused by personal properties of service delivery personnel?

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support from Technology Development Centre of Finland (Tekes). We thank our case companies for co-operation, access to information, and financial support.

REFERENCES

[1] J. Tiihonen and T. Soininen, 'Product Configurators – Information System Support for Configurable Products', *Tech. Report TKO-B137*, Helsinki U of Technology, Laboratory of Information Processing Science, 1997. Also in Richardson, Tom, ed. 1997, "Using Information Technology During the Sales Visit, Cambridge, UK: Hewson Group

[2] F. Salvador and C. Forza, 'Configuring products to address the customization-responsiveness squeeze: A survey of management issues and opportunities', *International Journal of Production Economics*, **91**, 273-291, (2004).

[3] P. Kotler, *'Marketing Management, 11th edition'*, Prentice-Hall, New Jersey, (2003).

[4] K-S. Paloheimo, I. Miettinen, and S. Brax, *Customer Oriented Industrial Services*, Report Series – Helsinki University of Technology, BIT Research Centre, Espoo Finland, (2004).

[5] D. Sabin and R. Weigel, 'Product Configuration Frameworks – A Survey', *IEEE Intelligent Systems & Their Applications*, **13**, 4, 42-49, (1998)

[6] G. da Silveira, D. Borenstein, and F.S. Fogliatto, 'Mass customization: Literature review and research directions', *International Journal of Production Economics*, **72,** 1-13, (2001).

[7] J. Harvey, L.A. Lefebvre, and E. Lefebvre, 'Flexibility and technology in services: a conceptual model', *International Journal of Operations & Production Management*, **17**, 1, 29-45, (1997).

[8] E.A. Papathanassiou, 'Mass customization: management approaches and internet opportunities in the financial sector in the UK', *International Journal of Information Management*, **24**, 387-399, (2004).

[9] H. Akkermans, Z. Baida, J. Gordijn, N. Peña, A. Altuna, and I. Laresgoiti, 'Value Webs: Using Ontologies to Bundle Real-World Services', *IEEE Intelligent Systems*, **19**, 57-66, (2004).

[10] A. Wimmer, J.I. Mehlau, and T. Klein, 'Object Oriented Product Meta-Model for the Financial Services Industry', *Proc. of the 2nd Interdisciplinary World Congress on Mass Customization and Personalization (MCPC'03)*, Münich, Germany, (2003).

[11] L. Peters, and H. Saidin, 'IT and the mass customization of services: the challenge of implementation', *International Journal of Information Management*, **20**, 103-119, (2000).

[12] R. Winter, 'Mass Customization and Beyond – Evolution of Customer Centricity in Financial Services'. *Workshop on Information Systems for Mass Customization (ISMC2001)*, Dubai, (2001).

[13] H. Meier, J.J. Schramm, and A. Werding, 'Development of a stage model based configurator to generate more customer-specific services and to support cooperative service networks', *3rd CIRP Int. Seminar on Intelligent Computation in Manufacturing Engineering (ICME2002)*, Ischia (Naples), Italy, 3-5 July, (2002).

[14] M. Dausch, and C. Hsu 'Mass-Customize Service Agreements for Heavy Industrial Equipment', *IEEE Int. Conf. on Systems, Man and Cybernetics*, **5**, 4809-4814, (2003).

[15] T. Böhmann, M. Junginger, and H. Krcmar, 'Modular service architectures: a concept and method for engineering IT services', *Proc. of the Int. Conf. on System Sciences (HICSS'03)*, 74-83, (2003).

[16] J. Chen, 'Improving reliability and speed in service mass customization: a case study in Chinese restaurant', *Proc. of Int. Conf. on Services Systems and Services Management (ICSSSM '05)*, **2**, 828-834, (2005).

[17] V.A Zeithaml, A. Parasuraman, L. Berry. 'Problems and strategies in services marketing', *Journal of Marketing*, **49**. 33-46, (1985).

[18] S. Kujala, *User Studies: A Practical Approach to User Involvement for Gathering User Needs and Requirements,* Acta Polytechnica Scandinavica, Mathematics and Computing Series No. 116, 2002

[19] A. Anderson, *Towards Tool-Supported Configuration of Services*. Master's Thesis, Helsinki University of Technology, Department of Computer Science and Engineering, 2005.

[20] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen, Towards a General Ontology of Configuration, *AI EDAM,* **12**, 357–372, (1998).

[21] M. Dumas, J. O'Sullivan, M. Heravizadeh, D. Edmond and A. Hofstede, 'Towards a semantic framework for service description', *Proc. of the 9th Int. Conf. on Database Semantics*, Hong-Kong, (2001).

[22] B. de Miranda. *'An Ontological Approach for the Use of Pricing Models to Sell Services.'* M.Sc. thesis, Department of Information Systems and Logistics, Free University, Amsterdam, The Netherlands

[23] J. Tiihonen, T. Soininen, I. Niemelä, and R. Sulonen, 'A Practical Tool for Mass-Customising Configurable Products', *Proc. of Int. Conf. on Engineering Design (ICED'03)*, Sweden, August 19-21, (2003).

[24] M. Stolze, S. Field, and P. Kleijer, 'Combining Configuration and Evaluation Mechanisms to Support the Selection of Modular Insurance Products', *Proc. of the 8th European Conf. on Information Systems, (ECIS 2000)*, Vienna, Austria. July 3-5, (2000).

[25] U. Junker, and D. Mailharro, 'Preference Programming: Advanced Problem Solving for Configuration', *AI EDAM*, **17**, 13-29, (2003).

[26] L. Ardissono, A. Felfernig, G. Friedrich, A. Goy, D. Jannach, G. Petrone, R. Schäfer, and M. Zanker, 'A Framework for the Development of Personalized, Distributed Web-Based Configuration Systems', *AI Magazine*, **24**, 93-108, (2003).

[27] M. Torrens, B. Faltings, and P. Pu, 'Smart Clients: Constraint Satisfaction as a Paradigm for Scaleable Intelligent Information Systems', *International Journal of Constraints*, **7**, 49-69, (2002).

[28] A. Goy, and D. Magro, 'STAR: a Smart Tourist Agenda Recommender', *ECAI 2004 Workshop on Configuration*, Valencia, Spain, August 23rd, (2004).

[29] M. Heiskala, J. Tiihonen, and T. Soininen, 'A Conceptual Model for Configurable Services', *IJCAI 2005 Workshop on Configuration*, Scotland, July 30, (2005).

[30] F. Bergenti, 'Product and Service Configuration for the Masses', *ECAI 2004 Workshop on Configuration*, Spain, August 23rd, (2004).

[31] Z. Baida, J. Gordijn, H. Sæle, A.Z. Morch, and H. Akkermans, 'Energy Services: A Case Study in Real-World Service Configuration', In: *A. Persson & J. Stirna (Eds.). Proc. of 16th Int. Conf. on Advanced Information Systems Engineering (CAiSE 2004),* Riga, Latvia, June 7-11, 2004. *LNCS*, **3084**, 36-50. Springer-Verlag, (2004).

[32] C.P. McLaughlin, 'Why variation reduction is not everything: a new paradigm for service operations', *International Journal of Service Industry Management*, **7,** 17-30, (1996).

[33] J. O'Sullivan, D. Edmond and A. Hofstede, 'What's in a Service? Towards Accurate Description of Non-Functional Service Properties', *Journal of Distributed and Parallel Databases*, **12**, 117-133, (2002).

[34] M. Heiskala, K-S. Paloheimo, and J. Tiihonen, 'Mass Customization of Services: Benefits and Challenges of Configurable Services', *Frontiers of e-Business Research (FeBR) 2005. Proc. of eBRF, 26.-28.9. 2005, Tampere, Finland*, (2006).

[35] T. Blecker, G. Friedrich, B. Kaluza, N. Abdelkafi, and G. Kreutler, *Information and Management Systems for Product Customization*. New York, USA: Springer, 2005.

Evolution of Configuration Models – a Focus on Correctness

Thorsten Krebs[1]

Abstract. Structure-based configuration models describe commonality and variability as well as restrictions within and between components of a product domain. Innovation in the development of products drives the development of new components and adaptation of existing components. As a consequence of evolution, the configuration model has to evolve in parallel with its referants in the world. A side effect of using configuration models for describing different and evolving products in one model is an increasing possibility for errors. Keeping an overview of the hundreds or thousands of configurable components and the increasing number of interrelations and restrictions for combinations of those is hard, if not impossible, without tool support. This paper focuses on how to keep a configuration model correct despite its changes. Change operations implement changes to the configuration model. After applying changes, syntactical correctness of the configuration model is checked with pre-defined invariants. A three-step process is introduced that consists of compiling change operations, propagating changes and validating the model after change propagation.

1 Introduction

Configuration tools are widely used to assemble new products out of a predefined set of existing components. The *product line* approach helps to configure similar but different products that together form a *product family*. These product family members are achieved by different compositions of the product components, with respect to technical possibilities and given customer requirements. In structure-based configuration, *configuration models* are used, which contain a textual description of the components, their capabilities, properties and relations to other components. Thus, configuration models describe commonality and variability as well as restrictions within and between components of a product domain. With this, all potentially configurable product family members are implicitly represented.

Reusability is widely identified as a key to improving (software) development productivity and quality. The reuse of (software) components results in fewer developments for a new system and in less time spent on development [2]. With reuse alone, however, innovation is not considered. New components are developed and existing components are evolved in order to derive new products. A consequence of evolution is that the configuration model has to be evolved in parallel with its referants in the world. Only then configuration tools can use the new knowledge. In such a dynamic environment the domain knowledge evolves continually [6].

Configuration models are input for configuration tools that offer automated support for correct and consistent products. In order to achieve this, configuration models have to be correct and consistent representations of the product domain. A side effect of continual evolution is an increasing possibility for errors because of the hundreds or thousands of configurable components in that model and the increasing number of interrelations and restrictions for combinations of those [19]. It is apparent that tool support can improve the evolution of configuration models. Modelling environments for creating, evolving and diagnosing configuration models have been addressed in recent years, but no one-fits-all solution is available.

This paper sketches the core ideas of a PhD thesis, which focuses on ensuring correctness of configuration models while applying changes. This paper describes work in progress and discusses goals reached so far as well as also future goals.

The remainder of this paper is organized as follows. Section 2 compares similar topics to see if existing ideas for modeling and reasoning about evolution of configuration models can be valuable. Section 3 introduces basic notions of evolution such as correctness of configuration models and change operations that implement the applicable changes. Section 4 details the application of change operations and validation of correctness. Section 5 discusses related work and Section 6 presents relevant topics for future work. Finally, Section 7 concludes this paper.

2 Lessons Learned

Looking beyond one's own nose, there are a number of related topics. This section takes a look at them to see what can be learned and used for evolution of structure-based configuration models.

2.1 Ontologies

Ontologies have gained popularity within the knowledge engineering community. Research in the area of ontology evolution quickens the pace as the semantic web gains interest. Generally, ontologies provide a "shared and common" understanding of a domain and facilitate "knowledge sharing and reuse" [5]. An ontology is an explicit specification of a *conceptualization* of the objects and other entities that are assumed to exist in some area of interest and the relationships that hold among them [7].

The shared and common understanding of an ontology is represented in a domain-independent vocabulary. Typically, frame-based languages are used to model ontologies. The central modeling primitives are concepts (known as frames) with properties (known as slots). Frames provide a context for modeling concepts in a taxonomy with slot-value pairs used to specify attributes of the concepts. Slots are often treated as objects that can be arranged in a hierarchy themselves.

[1] LKI, University of Hamburg, Germany, email: krebs@informatik.uni-hamburg.de

Configuration models use considerably more formalisms to represent product structures:

- compositional relations build a partonomy in which parts are related to aggregates,
- a cardinality specifies the amount of potential instances for compositional relations,
- there are optional and alternative concept definitions, and
- constraints represent restrictions between concepts and concepts properties.

Nonetheless, similar representation formalisms are used to represent objects in a taxonomic hierarchy. Some basic changes, like adding and deleting concepts or adding, deleting and modifying properties of concepts exist in both approaches. Hence, research on ontology evolution is considered to be of interest also for evolution of configuration models.

2.2 Knowledge Representation

A lot of effort has been put into formalizing configuration models over the last decades. Alternative formalisms for modeling product structures exist. This is apparent since a model is a *representation*: it imitates, resembles or stands for things that exist in the world [4, 15]. Most representation formalisms have a common basis. A configuration model uniquely identifies the components of a system, their properties, structure and possible variability. Modeling facilities of structure-based configuration models are:

Concepts represent products, components and other entities of the domain. Each concept $c \in \mathcal{C}$ carries a name which makes it uniquely identifyable within a domain. A concept may specify an arbitrary number of attributes. An attribute is a binary tuple that consists of a uniquely identifyable name within the specialization relation, and a value. The value of an attribute is restricted to a set of predefined value domains like integers, floats, strings, intervals of integers or floats and sets of all three. The set of attributes of a concept c is denoted with $\mathcal{A}_c$.

Specialization relations relate a concept c_1 to its subconcepts c_2 and with this form a taxonomic hierarchy $(c_1, c_2) \in \mathcal{H}$. Every concept has exactly one ancestor and can have an arbitrary number of descendants. Multiple inheritance is explicitly ruled out with the definition of a tree structure. The specialization relation is transitive. Therefore, the superconcept of a superconcept of a concept c is an *indirect* superconcept of c. The set of relations in which concept c is the aggregate is denoted with $\mathcal{R}_c$.

Compositional relations relate a part p to its aggregate a and with this form a partonomy $(a, p) \in \mathcal{P}$. Objects are either *primitive* or *composite*, i.e. they reside at the leaves of a component hierarchy or are the root of a subgraph, respectively. One aggregate can have multiple parts in compositional relations.

Constraints express interdependencies and restrictions between concept definitions and their properties such as incompatibilities, attribute values that depend on other attributes (of other concepts), and so on. There are local and global constraints $L \cup G = \Gamma$. *Local constraints* are applied to single attributes, a single concept or to a relation between two concepts. *Global constraints* involve a larger number of concepts and attributes or relations.

Usually, this is a static structure that does not take versioning into account. It can be represented by a graph whose nodes and edges correspond to concepts and relationships, respectively [3]. For modeling product families (i.e. modeling a set of products within one model), this static, non-variable representation is enriched with notions to describe variability; like optional and alternative definitions.

Evolution of knowledge strutcures that are represented by concepts and relations is a well-understood domain. Usually, evolution is divided into historical and logical versioning – i.e. a two-dimensional representation in time and space, respectively. Typically, *versions* are characterized as "descendants of some existing version, if not the first one, and can serve as an ancestor for multiple versions" [8].

There are numerous problems within the domain of evolution and versioning. However, this paper focuses on correctness of configuration models and considers versions of models more appropriate. A change transforms a configuration model $\mathcal{M}$ into a new configuration model $\mathcal{M}'$. A *valid* change c, in addition, is a function that transforms a correct configuration model into another correct configuration model $c : \mathcal{M} \mapsto \mathcal{M}'$. Changed configuration models are different from each other but still represent the same underlying product domain. They are different versions of the same model. A version captures a specific point in time. This means that different versions describe different sets of configurable products – according to the existing configurable components at that time.

2.3 Configuration Management

Configuration management (CM) systems are concerned with managing the evolution of large and complex systems represented in an explicit, unambiguous configuration model [23, 24]. CM serves management support (i.e. controlling changes to products) as well as development support (i.e. providing functions which assist developers in performing coordinated changes to products) [3].

Basic requirements for configuration management are version control (keeping track of changes to components, supporting parallel development and enabling branching and merging), build management (the process of building components and producing a "bill-of-materials"), and process control (a set of policies, including monitoring changes, notification on changes, access control and reporting) [14].

Typically, CM systems distinguish between the *product space* and the *version space*. The product space consists of the configurable objects and their relationships, while the version space organizes the versions of these objects. Versions are organized into a derivation history [8]. Key features of versioning are the organization of the version space, the interrelations between product space and version space, and operations for retrieving existing versions and constructing new ones.

Traditional configuration models capture the versioning in space. This means they represent all admissible configurations. Evolution, i.e. versioning in space and time, is typically not supported [11]. Methods from CM can help at this point.

3 Evolution

One of the key benefits of configuration mechanisms is to guarantee consistency and correctness of configured products. To reach this goal, the configuration model has to be consistent and correct. In a dynamic environment with continual changes to the configuration model it is apparent that a major goal of the evolution process is keeping the model consistent and correct despite its changes.

Consistency means that none of the facts deducible from a configuration model contradict one another. Thus, consistency can be considered as an agreement among the knowledge entities with respect to

the semantic of the underlying modeling language [20]. *Correctness* of a configuration model is asserted when it is correct with respect to the underlying modeling language.

Therefore, two levels of analyzing configuration models can be distinguished: a *syntactic* and a *semantic* level. This section deals with the syntactic analysis of configuration models. As a first step towards an evolution process that keeps configuration models consistent and correct, the following subsections discuss *well-formed* (i.e. syntactically correct) configuration models and introduce invariants to check the model against.

3.1 Correctness

A configuration model $\mathcal{M}$ is *well-formed* with respect to a set of syntax invariants $\mathcal{I}$ if for all $i \in \mathcal{I}$, $\mathcal{M}$ satisfies the invariant $i(\mathcal{M})$.

Invariants are conditions that must hold for every configuration model [1]. Every change applied to a configuration model must maintain the correctness defined by the invariants. Three syntax invariants that are needed to understand the upcoming example are introduced in the following.

I1 - Specialization Relation Invariant A concept has exactly one superconcept; if not the root concept.

$$\forall c_1 \in \mathcal{C} \setminus \{\text{root}\}\ \exists c_2 \in \mathcal{C} \text{ such that } (c_2, c_1) \in \mathcal{H}$$
$$\text{and } \forall c_3 \in \mathcal{C} \text{ with } (c_3, c_1) \in \mathcal{H} \Rightarrow c_2 = c_3$$

I2 - Inheritance Invariant The value of every attribute a of a concept c_1 that is inherited from c_2 has to be a subset of the corresponding value of a' of c_2.

$$\forall c_1, c_2 \in \mathcal{C} \text{ with } (c_2, c_1) \in \mathcal{H}, \forall a' \in \mathcal{A}_{c_2}\ \exists a \in \mathcal{A}_{c_1}$$
$$\text{such that name}(a) = \text{name}(a') \wedge \text{value}(a) \subseteq \text{value}(a')$$

I3 - Composition Reference Invariant All parts that are referenced in a compositional relation have to be defined as concepts.

$$\forall (a, p) \in \mathcal{P}\ \exists c \in \mathcal{C} \text{ such that name}(p) = \text{name}(c)$$

Of course, these are just simple examples of invariants. A lot more invariants need to be defined for guaranteeing well-formed configuration models.

3.2 Example

A PC consists, among others, of a VGA card. Two types of VGA cards are shown in Figure 1, AGP cards and the new type, PCIe. Because PCIe is the new type, AGP cards are no longer produced and should be removed from the configuration model. This is indicated by the large "X" in place of where the AGP card is in the hierarchy.

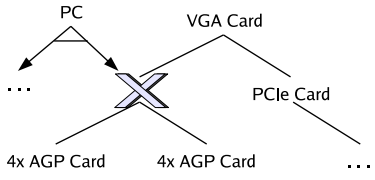


Figure 1. Extract of the PC domain. The taxonomic hierarchy of VGA cards is shown for two types: AGP and PCIe.

The following discussions focus on two issues of this removal. The first issue is concerned with the 4x AGP and the 8x AGP cards that are specializations of the AGP card. The second issue is concerned with the AGP card being a part of a PC.

3.3 Change Operations

Change operations have to be clearly defined. They must compare and present *structural* and *semantic* changes rather than *syntactical* changes in text representation. The latter is e.g. common when comparing versions of software code. But the same content can be modeled with different syntactical means or simply in different order. Two configuration models can be the same conceptually, but have very different text representations [18].

Base operations represent elementary changes that cannot be decomposed into simpler changes. Base operations are for example adding or deleting a concept definition, adding and deleting a concept attribute or a compositional relation, etc. This fine granularity of separated changes is not always appropriate. Changes should also be defined on a higher level that allows semantic interpretation. *Compound operations* group base operations into a meaningful unity [10]. Compound operations are for example modifications to compositional relations or constraints, that affect multiple concepts, or *tree-level* changes such moving a subtree of concepts or modifying an attribute value, which affects all descendants as well, due to inheritance within the taxonomy.

Grouping base operations to more meaningful compound operations is reasonable due to the fact that one (compound) operation is more concise than multiple (base) operations that might be needed because a change implies action in different places of the configuration model. This makes the use of compound operations more suitable for a user interface. And last but not least, they are needed because elementary changes may lead to incorrect configuration models [22]. In this case, additional change operations should re-transform the configuration model into a correct state [13].

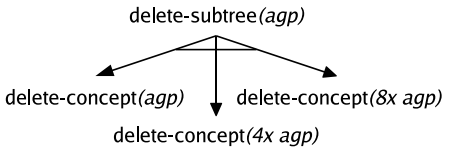


Figure 2. Definition of the compound operation delete-subtree(apg). The arrows indicate its composition.

An example for a compound change operation is deleting a subtree of concept definitions. Figure 2 depicts how this operation delete-subtree(apg) is composed of deleting concept agp (delete-concept(agp)) and recursively deleting its descendants $4\text{x}agp, 8\text{x}agp \in \mathcal{C}$ with $(agp, 4\text{x}agp), (agp, 8\text{x}agp) \in \mathcal{H}$ (delete-concept$(4\text{x}agp)$ and delete-concept$(8\text{x}agp)$).

Compound operations can be composed of base operations and possibly other compound operations. The compositions forms a tree structure in which compound operations have further descendants and base operations are the leaf nodes, respectively.

Change operations have *preconditions*. This means that they can only be applied if certain assertions are satisfied by the configuration model. Typical preconditions are that some knowledge entity exists – i.e. it is defined in the configuration model. Deleting a concept c for example can only be applied if c is modeled: $c \in \mathcal{C}$.

The application of a change operation also has *postconditions* on the model: it changes assertions, introduces or deletes assertions. The set of assertions that is satisfied by the configuration model can increase or decrease, but definitely changes when an operation is applied. Adding elements to the configuration model usually increases the set of assertions while removing elements usually decreases it. Deleting concept c for example removes the assertion that c is modeled: $c \notin \mathcal{C}$.

Preconditions and postconditions both are represented by propositions that describe the content of assertions in a way that they are either true or false. A proposition $p \in P$ can define the existence (e.g. $c \in \mathcal{C}$) or absence (e.g. $c \notin \mathcal{C}$) of any knowledge entity as well as specific values for attributes, relations and constraints (e.g. value$(a) = 1$).

By organizing base operations in a taxonomy, the inheritance mechanism can be exploited to specify common properties of the operations in an efficient way. Common and varying properties of base operations are the preconditions defining their applicability and the postconditions they have on the configuration model. A taxonomy of change operations for ontology evolution is for example given in [9].

A change operation is *sound* if the operation itself is applicable and all operations it contains are applicable in some order. All sound changes to manipulate a configuration model can be specified by base or compound operations. Changes can be applied to a correct version of the model $\mathcal{M}$, and after all operations are performed, the model must transform into another correct version $\mathcal{M}'$ [22].

However, simply concatenating base operations to a compound operation in a pre-defined manner has some drawbacks:

- There may be a mismatch between the intent of the change and the way the pre-defined operation is composed.
- Unnecessary changes may be performed if they are applied independent from each other.
- The applicability of change operations depends on what is currently modeled. Therefore, the order in which base operations inside a compound operation are applicable can vary.

The first two items have also been identified by [21]. The third item introduces the fact that the knowledge inside the configuration model dictates when certain operations are applicable and when they can not. The following section details the dynamic composition of base operations into a compound operation.

4 Evolution Process

In the ideal case, a change operation can simply be applied to a configuration model. But two issues make this process a bit more complex. These issues are

1. that change operations are not always applicable – depending on the model, and
2. the interrelations of concept definitions according to taxonomic and compositional relations and constraints.

The latter means that a change operation can have unforeseen consequences leading to an incorrect configuration model, which has to be resolved by additional changes.

One change to the configuration model is implemented with one change operation. The issues mentioned above show the necessity of arranging the evolution process in three steps:

Compilation of Complex Change Operations: Complex change operations are compiled, based on the preconditions and postconditions of the corresponding base operations, before this operation is applied. A particular change may be implemented with different operations, depending on the configuration model.

Change Propagation: The identified change operations are applied to the configuration model with respect to their applicability. Specific combinations of changes and the affected knowledge entities require an analysis of the model for additional change operations.

Change Validation: After a change to the configuration model has been propagated, all defined syntax invariants are checked against the model. When an incorrectness is detected, additional change operations have to be identified to implement changes resolving this incorrectness. This means that the previous steps are performed again – leading to an iterative process.

The following three sections detail the compilation of complex change operations, the change propagation and the change validation, respectively.

4.1 Compilation of Complex Change Operations

Applying changes to a configuration model modifies its contents and / or structure. This is wanted since evolution intends to change the model. However, some changes can have unforeseen consequences. These can arise because of constraints and other, change-dependent, interrelations. Changing concept attributes for example also affects all descendants – see the inheritance invariant (**I2**).

Consequences of change operations can be evaluated by analyzing the configuration model. Additional changes that become necessary for repairing an incorrect model can be identified based on information about the nature of the change and the affected knowledge entities.

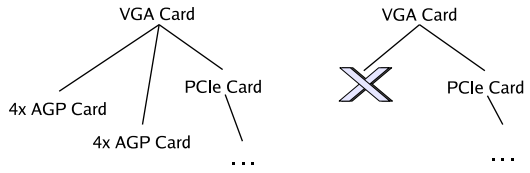


Figure 3. The concept VGA card is deleted. In the left its subconcepts are kept while in the right they are also deleted.

Deleting the AGP card, for example, is a simple operation when this concept is a leaf node.[2] Because there are descendants $4\text{x}agp, 8\text{x}agp \in \mathcal{C}$ with $(agp, 4\text{x}agp), (agp, 8\text{x}agp) \in \mathcal{H}$, however, agp cannot be simply deleted. This would violate the specialization relation invariant (**I1**). There are two possible resolutions for this incorrectness: the 4x agp and 8x agp are also deleted (see Figure 3 on the right), or they are moved to some new parent. Considering the semantics of inheritance, vga, the parent of agp, is the next indirect superconcept and should be used as a new superconcept. This is shown in Figure 3 on the left. Another violation is that of the composition reference invariant (**I3**) because agp is referenced in a compositional relation $(pc, agp) \in \mathcal{P}$. In this case the compositional relation should also be deleted.

Both violations of invariants and the additional change operations identified to repair them are given in Table 1.

Note that in general there are two possible changes to repair the violated composition reference invariant (see Table 1). These are

[2] For reasons of simplicity, at this point only dependencies concerning specialization and compositional relations are discussed, not considering constraints.

Change	Incorrectness	Additional Changes
Remove Concept	$\exists c_2 \in \mathcal{C}$ such that $\forall c_1 \in \mathcal{C},\ (c_1, c_2) \notin \mathcal{H}$	
		add-specialization(c_1, c_2)
		delete-concept(c_2)
	$\exists (a, p) \in \mathcal{P} \wedge \neg \exists c \in \mathcal{C}$ with name$(p) =$ name(c)	
		add-concept(c)
		delete-composition(a, p)

Table 1. Identifying additional change operations.

adding the corresponding concept and deleting the relation. Both are meaningful to repair the incorrectness – depending on the nature of change that led to this incorrectness. Adding the concept that is missing for the compositional relation, however, is not appropriate in the case that it has been deleted before due to the fact that this inverts the intended deletion.

Change operations that are inappropriate for repair can generally be identified based in their preconditions and postconditions. Deleting a concept c has the precondition $c \in \mathcal{C}$ and postcondition $c \notin \mathcal{C}$, while deleting concept c has the exactly inverse precondition $c \notin \mathcal{C}$ and postcondition $c \in \mathcal{C}$. It is apparent that either of these changes is not appropriate to repair an incorrectness that occurred because the other change.

Not every incorrectness can be resolved automatically. Some alternative resolutions can seem equally well suited. Figure 3 for example shows two conceivable scenarios. The descendants of the AGP card do not have to be deleted; they may be kept for legacy support.

4.2 Change Propagation

The preconditions and postconditions of change operations can be used to compute a temporal order. For some base operations there might not be any limitations while further operations can only be applied after postconditions of others satisfy preconditions of these.

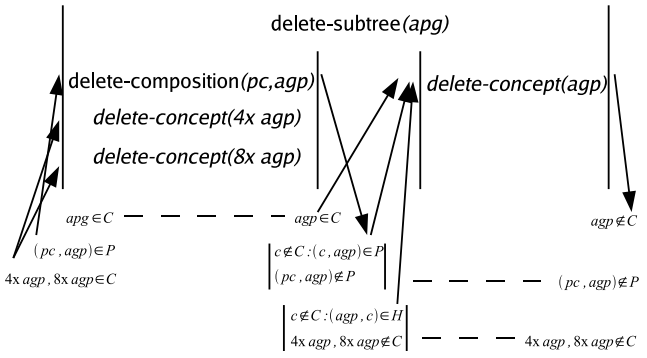


Figure 4. Temporal order of the compound operation delete-subtree(agp). Time proceeds from left to right. Arrows represent pre- and postconditions. Dashed lines indicate persistence of propositions, if not negated.

Figure 4 depicts the temporal order of operations inside delete-subtree(agp). The descendants of agp and the compositional relation between pc and agp are deleted first (delete-subtree$(4\text{x}agp)$, delete-subtree$(8\text{x}agp)$ and delete-composition(pc, agp)). Preconditions for these are their existence – that is $4\text{x}agp, 8\text{x}agp \in \mathcal{C}$ such that $(agp, 4\text{x}agp), (agp, 8\text{x}agp) \in \mathcal{H}$ and $(pc, agp) \in \mathcal{P}$. After that, agp itself can be deleted (delete-concept(agp)). Preconditions for this are the existence $agp \in \mathcal{C}$ and that there are no descendants $d \notin \mathcal{C}$ such that $(agp, d) \in \mathcal{H}$ and no compositions $c \notin \mathcal{C}$ such that $(c, agp) \in \mathcal{P}$.

The existence preconditions ($agp \in \mathcal{C}$ and $4\text{x}agp, 8\text{x}agp \in \mathcal{C}$ such that $(agp, 4\text{x}agp), (agp, 8\text{x}agp) \in \mathcal{H}$) have to be satisfied before the compound operation can be applied. Therefore, they are also preconditions of this compound operation. This is different for the absence preconditions ($c \notin \mathcal{C}$ such that $(agp, c) \in \mathcal{H}$ and $c \notin \mathcal{C}$ such that $(c, agp) \in \mathcal{P}$): they are postconditions of other operations inside this compound operation and therefore do not have to be satisfied beforehand.

Note that in Figure 4 the propositions $(pc, agp) \notin \mathcal{P}$ and $c \notin \mathcal{C}$ such that $(a, agp) \in \mathcal{P}$ are treated equally because the PC is the only aggregate that the AGP card is part of. Analogously, the propositions $4\text{x}agp, 8\text{x}agp \notin \mathcal{C}$ and $c \notin \mathcal{C}$ such that $(agp, c) \in \mathcal{H}$ are also treated equally.

At least one operation within a sound compound operation must be applicable! If there are base operations that do not have preconditions or that only have preconditions satisfied by the configuration model, these can be applied first; in an arbitrary order. After a change operation has been applied, the assertions represented by the configuration model have changed. This means that preconditions for other base operations may have become satisfied. Therefore, the applicability of all operations has to be verified after each application of an operation.

The algorithm to compute a temporal order between the base operations B within a sound compound operation o is given in the following. The configuration model is denoted with $\mathcal{M}$.

ALGORITHM: computeTemporalOrder(o)

1. Initialize the set of applicable operations $A = \emptyset$.
2. If $B = \emptyset$, then return A.
3. For all $b \in B$ do:
   (a) If the set of preconditions of b is empty ($P_b = \emptyset$), then
      i. add b to the set of applicable operations ($A = A \cup \{b\}$)
      ii. remove b ($B = B \setminus \{b\}$).
   (b) Else, for all $p \in P_b$, do:
      i. If p is not satisfied ($\mathcal{M} \cap p = \emptyset$), then
         - move b to the end of B ($\{b, b_1, \ldots, b_n\} \rightarrow \{b_1, \ldots, b_n, b\}$)
         - continue with next b.
      ii. Else if p is the last element of P_b, then
         - add b to the set of applicable operations ($A = A \cup \{b\}$)
         - remove b ($B = B \setminus \{b\}$).
         - continue with next b.
      iii. Else continue with next p.
4. Continue with step (2).

In principle, the set of operations that belong to a compound operation can be split into two sets, according to whether they are applicable or not. An operation is applicable if all its preconditions are satisfied, if any, and cannot be applied elsewise. The order in which applicable operations are applied is not of importance.

4.3 Change Validation

After change propagation has taken place, all invariants defined for the configuration have to be checked against the model. If no violation is detected, the configuration model is transformed into a new,

correct, state – i.e. into a new version of that model. If a violation of an invariant is detected, this means that the change has introduced an incorrectness.

Incorrect configuration models are not viable. There are two possibilities to cope with this:

1. the configuration model has to be repaired by applying additional changes, or
2. the intended change has to be undone.

The latter indicates the need to define transaction sets for the the changes to a configuration model. All changes are accepted in one go if the new version of the configuration model is correct. If the new version is not correct and no additional changes are intended, it is always possible to reclaim the version from before the change.

5 Related Work

There are a few research groups also dedicated to the evolution of structure-based configuration models and models for product families. For example, in [17, 16] the notion of generic objects and the division of versions into variants and revisions [8] is used for evolution of configuration knowledge. A lot of ideas also come from previous work, for example [12, 13].

Ontology evolution has gained interest in recent years. This may have a lot to do with the Semantic Web. In [9] for example there is a taxonomy of change operations defined for ontology evolution. [20] focuses on consistency of evolving ontologies and defines invariants to check consistency of an ontology.

6 Outlook

This paper describes work in progress. This means that some work still has to be done. Future work includes the following issues:

- Invariants define well-formed configuration models. A list of invariants that completely cover all facilities of the modeling language is needed in order to guarantee correctness of a model despite its changes.
- Change operations are to be defined in a taxonomy. The characteristics according to which they may be aligned have to be identified. Possible choices are the types of changes (add, delete, modify), the knowledge entities they operate on (concept, specialization relation, compositional relation, constraint) or the preconditions defining their applicability and the postconditions the operations have on the configuration model.
- Changes applied to a configuration model may potentially violate invariants. Different types of changes applied to different knowledge entities are conceivable (see Section 4.3). The identification of appropriate change operations for all cases is still an open issue. Possible choices of characteristics for this identification are the arguments of an operation, its preconditions and postconditions, or a semantic interpretation of the nature of the change.

7 Conclusion

This paper addresses potential problems that may arise within continual evolution of configuration models. It presents an approach to prevent incorrect configuration models. This approach consists of a set of invariants to check the syntactical correctness of model, clear semantics of changes to the configuration model and their implementation in change operations. A three-step evolution process defines how to compile compound change operations, propagate changes and validate the configuration model after change propagation.

REFERENCES

[1] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth, 'Semantics and implementation of schema evolution in object-oriented databases', in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pp. 311–322. ACM Press, (1987).

[2] Ted J. Biggerstaff and Charles Richter, 'Reusability framework, assessment, and diretions.', *IEEE Software*, **4**(2), 41–49, (1987).

[3] Reidar Conradi and Bernhard Westfechtel, 'Version models for software configuration management', *ACM Computing Surveys (CSUR) archive*, **30**(2), 232–282, (1998).

[4] Randall Davis, Howard E. Shrobe, and Peter Szolovits, 'What is a knowledge representation?', *AI Magazine*, **14**(1), 17–33, (1993).

[5] Dieter Fensel, *Ontologies – A Silver Bullet for Knowledge Management and Electronic Commerce*, Springer Verlag, 2001.

[6] Dieter Fensel, 'Ontologies: Dynamics networks of meaning', in *Proceedings of the 1st Semantic web working symposium*, (2001).

[7] Thomas R. Gruber, 'Ontolingua: A mechanism to support portable ontologies', Technical Report KSL 91-66, Version 3.0, Stanford University, Knowledge Systems Laboratory, (1992).

[8] Randy H. Katz, Ellis E. Chang, and Rajiv Bhateja, 'Version modeling concepts for computer-aided design databases', in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pp. 379–386. ACM Press, (1986).

[9] Michel Klein, *Change Management for Distributed Ontologies*, Ph.D. dissertation, Vrije Universiteit Amsterdam, 2004.

[10] Michel Klein and Natalya Noy, 'A component-based framework for ontology evolution', Technical Report IR-504, Department of Computer Science, Vrije Universiteit Amsterdam, (2003).

[11] Tero Kojo, Tomi Männistö, and Timo Soininen, 'Towards intelligent support for managing evolution of configurable software product families', in *Proceedings of 11th International Workshop on Software Configuration Management (SCM-11)*, pp. 86–101. Springer Verlag, (2003).

[12] Thorsten Krebs, Lothar Hotz, Christoph Ranze, and Guido Vehring, 'Towards evolving configuration models', in *PuK2003 – Papers from the KI Workshop*, pp. 123–134, (2003).

[13] Thorsten Krebs, Katharina Wolter, and Lothar Hotz, 'Mass customization for evolving product families', in *Proceedings of International Conference on Economic, Technical and Organizational Apects of Product Configuration Systems*, pp. 79–86, (2004).

[14] David B. Leblang and Paul H. Levine, 'Software configuration management: Why is it needed and what should it do?', in *Selected papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*, pp. 53–60. Springer-Verlag, (1995).

[15] Mark H. Lee, 'On models, modelling and the distinctive nature of model-based reasoning', *AI Communications*, **12**(3), 127–137, (1999).

[16] Tomi Männistö, *A Conceptual Modelling Approach to Product Families and Their Evolution*, Ph.D. dissertation, 2000.

[17] Tomi Männistö, Hannu Peltonen, and Reijo Sulonen, 'View to product configuration knowledge modelling and evolution', in *Configuration – Papers from the 1996 Fall Symposium*, pp. 111–118. AAAI Press, (1996).

[18] Natalya F. Noy, Sandhya Kunnatur, Michel Klein, and Mark A. Musen, 'Tracking changes during ontology evolution', 259–273, (2004).

[19] Daniel Sabin and Rainer Weigel, 'Product configuration frameworks – a survey', *IEEE Intelligent Systems*, **13**(4), 42–49, (1998).

[20] Ljiljana Stojanovic, *Methods and Tools for Ontology Evolution*, Ph.D. dissertation, Universität Karlsruhe, 2004.

[21] Ljiljana Stojanovic, Alexander Maedche, Boris Motik, and Nenad Stojanovic, 'User-driven ontology evolution management', in *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management*, (2002.).

[22] Ljiljana Stojanovic and Boris Motik. Ontology evolution within ontology editors, 2002.

[23] W. F. Tichy, 'Tools for software configuration management', in *Proceedings of the International Workshop on Software Version and Configuration Control*, pp. 1–20. Teubner Verlag, (1988).

[24] David Whitgift, *Methods and Tools for Software Configuration Management*, Wiley & Sons Ltd., 1991.

Short Papers and Position Statements

Turning a Configurator into a Bargaining Table

Songlin Chen and **Mitchell Tseng** [1]

Abstract. Customers are often unable to precisely articulate their requirements *a priori*. Hence requirements are negotiable and should be negotiated effectively during product configuration. This paper views product configuration as a process of negotiation and presents a framework, problem formulation, and problem solving procedure. The motivation is to integrate the functionalities of a configurator with a Negotiation Support System.

1 INTRODUCTION

Product configurators have been widely used for quotation making by companies that offer custom made products like industrial machinery, commercial refrigeration systems, etc. Significant reduction in quotation time and cost, improvement in quotation accuracy and customer satisfaction have been reported ([1]). However, some companies find themselves in a difficult situation: their configurators are best in breed and maintained up to date, but are idle most of the time while 'special teams' of experienced engineers are frequently called in for RFQ processing. As the engineering director of an escalator manufacturer described the problem, "All these tools (configurators) are working fine if the sales input is strictly according to the offering. If there are customer wishes outside of the offering or other ambiguities, the effectiveness of such tools becomes less and less and process becomes more manual." What's disturbing is that such outlier customer requirements seem to be the norm instead of exception. Despite the billions of product variants offered, the company found that less than 30% of the quotes were generated directly from the configurators.

Why are customer requirements so *irregular*? More importantly, how can we make configurators more robust to effectively handle them? These are the questions that this paper aims to address.

2 CUSTOMER REQUIREMENT NOISE

Several factors contribute to the irregularity of customer requirements. The first is product complexity. Many attributes need to be specified to fully describe a product and there are constraints between different attributes, which customers may not be aware of. The second is tradeoff making. Customers often have to make tradeoffs among multiple competing objectives (e.g. price vs. performance). More choices mean more efforts for comparison, and too many choices may lead to confusion ([2]). The third factor is the so-called stickiness of need information. According to von Hippel, need information for custom products is generally *sticky*, i.e. costly to transfer ([3]). The fourth factor is the information asymmetry between customers and suppliers. Customers often don't have a clear understanding of what is available or feasible when they specify requirements.

Given these factors, it is unlikely that customer requirements will be 'strictly according to the offering' of a product configurator. So from a supplier's point of view, customer requirements are signal of needs with noises. The task of configuration has conventionally been defined as selecting and combining parts to satisfy *given* specifications ([4]). Configurator design implicitly assumes an open-loop typology, which is vulnerable to noises. If noises are not filtered, the configurator will either fail to find a solution or find a solution that is suboptimal.

One approach to remove requirement noises is to provide a template for customers to specify their requirements via sequentially answering a series of questions. The advantage of this approach is that requirement consistency and completeness will be guaranteed. The disadvantage is that there may be too many questions to be answered, particularly for complex products. Another disadvantage is about the sequence of the questions. Although the sequence generally follows a hierarchical structure, it is often difficult and tedious to backtrack and revise a previous choice. A practical limitation of this approach is that customers may not be willing to use the requirement template because they normally source from several competing suppliers.

3 TURNING A CONFIGURATOR INTO A BARGAINING TABLE

3.1 Configuration via Negotiation

There will always be customer requirements outside of a configurator's offerings. It's important to recognize that some of these irregular requirements genuinely represent customers' real needs; some are simply distorted need information. In the former case, the company needs to expand its offerings. In the latter case, the company needs a better mechanism to explore available offerings and discover customers' true needs. Blindly expanding offerings in this case may further confuse customers and distort the need information. It's the latter case that this study is focused upon. [2]

It's also important to recognize that configuration is essentially about matching customers' needs with suppliers' capabilities. Mismatch results mainly from the information asymmetry and preferential difference between customers and suppliers. Negotiation, the natural discourse of give and take between buyers and sellers, provides a basis of interaction for both sides to acquire information and to deal with conflicts. Configuration can be taken as a process of negotiation, in which customers and suppliers exchange information of needs and capabilities, jointly resolve preferential conflicts, and collaboratively explore alternatives.

Approaching configuration via negotiation is actually widely practiced during customer/sales interaction. The process works as the fol-

[1] Advanced Manufacturing Institute, The Hong Kong University of Science & Technology, Hong Kong, email: {songlin, tseng}@ust.hk

[2] The former case is a special case of the latter, in which customers know exactly what they want.

lowing: sales representatives will review customers' initial requirements, then empirically modify certain attributes if no feasible configuration is available; the modified requirements will be loaded into the configurator; the resulted configuration will then be presented to the customer, who will update his requirements if not satisfied. The process repeats until a satisfactory configuration is found or the configuration task is transferred to a 'special team' for exception handling. Figure 1 depicts the information flow of this process.

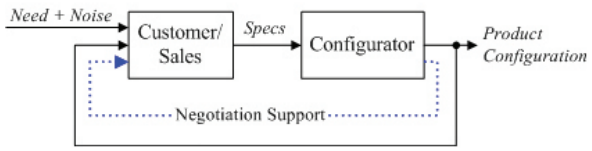


Figure 1. Information Flow

A drawback with current practice is the lack of knowledge support and over-reliance on human experience. The only feedback from the configurator is the resulted configuration, which is often difficult to interpret or improve given the vast options available. With bounded rationality, sales representatives tend to develop several 'typical configurations' and get anchored to them over time. As a result, part of the variety offered the company is not accessible ([5]). This may also explain why configurators are not extensively utilized for quotation making. The motivation of this paper is to integrate negotiation support functionalities with a configurator so as to support customer/sales interaction systematically.

3.2 Negotiation Support

In the past two decades, there's been extensive research in designing information systems to facilitate formulation and resolution of negotiation problems. Among these so-called e-negotiation systems, Negotiation Support Systems (NSS) have been proved to be effective in helping negotiators to realize joint gains and improving negotiation efficiency ([6]). NSS functionalities range from facilitating negotiation preparation to mediation, offer evaluation and post negotiation settlement etc.

This study focuses on the process of interactive configuration problem solving. Negotiation support is focused on leveraging a configurator's product knowledge to provide additional feedback to facilitate the customer/sales interaction (the dotted line in figure 1). *What* and *how* to feedback are the primary questions to be addressed.

3.3 A Negotiation Framework

Configuration is a special form of design ([4]). Design, according to the axiomatic design principles of Nam Suh, is a series of mappings from customer variables (CVs) to functional requirements (FRs), to design parameters (DPs), and to process variables (PVs) ([7]). One feature of configuration as design is that it's not creative in nature, which basically means the mappings from FRs to DPs to PVs have already been established[3] and the task of configuration is to determine the specific values.

[3] {FR} = [A]{DP} = [A][B]{PV}, where [A] and [B] are design matrices that indicate the mapping relationships, which could be discrete or continuous, linear or nonlinear.

Raiffa et al. define negotiation as a process of joint decision making, which entails joint consequences, or payoffs, for each individual ([8]). Based on this definition, there are two elements for a negotiation to take place: first, a channel for communication through which decisions can be made jointly; second, a mechanism for each individual to evaluate the consequences so that alternatives can be compared and negotiation can move forward. Based on this proposition, a negotiation framework for configuration is developed (Figure 3).

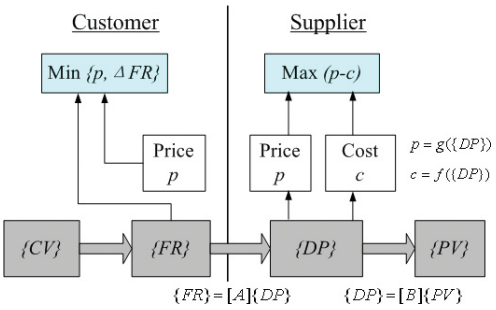


Figure 2. A Negotiation Framework for Configuration

The mapping relationships between FRs, DPs, and PVs establish a channel for communication. As payoff evaluation is concerned, generally speaking, customers aim to have the best product with lowest price and best delivery terms, while suppliers aim to maximize profit[4].

3.4 Problem Formulation

Product configuration can be formulated as a mixed integer programming(MIP) problem ([9]), which can be converted into an integer goal programming problem ([10]). The decision problem faced with customers and suppliers are formulated as in figure 3 and 4 respectively.

Given:

Target values for FR_i: G_i

Relative importance of price: α

Relative importance of FR_i: w_i

Find:

$\{FR\}, p$

Satisfy:

$$FR_i + d_i^- - d_i^+ = G_i$$

$$d_i^- * d_i^+ = 0, d_i^- \geq 0, d_i^+ \geq 0$$

Bounds: $\underline{FR_i} \leq FR_i \leq \overline{FR_i}$

Constraints: $C_f(FR) \leq 0$

Reservation price constraint: $p < p_0$

Minimize:

$$Z_c = \alpha p + (1-\alpha)\sum w_i(d_i^- + d_i^+)$$

Figure 3. Customer's Information and Decision

[4] Without loss of generality, price is assumed to be the only non-technical attribute, and PVs are omitted from discussion because they are functionally equivalent to DPs.

d_i^+ and d_i^- measure the over- and under-achievement of the goals respectively. They provide a direct measure of the mismatch between customer requirements and suppliers' offerings.

Given:

$\{FR\} = [A]\{DP\};$

Find:

$\{DP\}, p$

Satisfy:

Bounds: $\underline{DP_i} \le DP_i \le \overline{DP_i};$

Constraints: $C_d(\{DP\}) \le 0;$

Costing and pricing rules:

$c = f(\{DP\}); p = g(\{DP\})$

$p > c$

Maximize:

$Z_s = p - c$

Figure 4. Supplier's Information and Decision

Goals are desirable to achieve but do not have to be satisfied. The solution of a goal programming problem is the alternative with minimum deviations from the goals. Goals can be prioritized into different levels or assigned different weights to indicate their relative importance. Such a formulation is realistic and provides the necessary flexibility for dealing with irregular customer requirements. Even when customer requirements are 'not strictly according to the offering', solving the problem is still able to generate an intermediate solution to move negotiation forward.

Since the need information ({FR}) and solution information ({DP}) are distributed within the customer and supplier respectively, solving the configuration problem is interactive in nature. Interactive goal programming methods have been applied for financial planning([11]). They can be adapted for configuration problem solving. A general problem solving procedure is shown in figure 5.

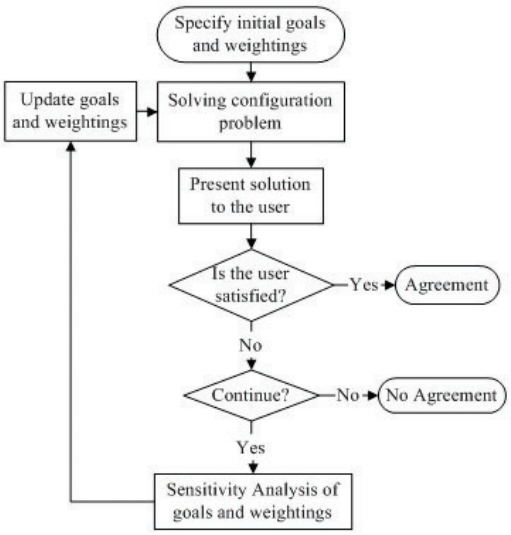


Figure 5. Problem Solving Procedure

4 SUMMARY

Because of the presence of multiple attributes, objectives, and suppliers, customer requirements for customized complex products often do not fit right into the offerings of a supplier's configurator. Many product configurators respond poorly to such *irregular* requirements because requirements are assumed as given or there's no effective feedback to the user. This paper treats customer requirements as negotiable and views configuration as a process of negotiation. A negotiation framework for configuration is developed, and an interactive goal programming method is proposed for configuration problem solving. The objective is to convert a product configurator into a bargaining table, based on which customer needs and supplier capabilities can be systematically explored and matched. Work is currently in progress on algorithm and prototype system development. Future work is needed on developing efficient methods for sensitivity analysis in mixed integer goal programming. The convergence property of the negotiation process also needs further investigation.

ACKNOWLEDGEMENTS

We would like to thank the Research Grant Council(RGC) of HKSAR (China) and Natural Science Foundation of China(NSFC) for their support under "The theory, methods and key technology of production organization and management for mass customization".

REFERENCES

[1] Hvam, L. and S. Pape, 'Optimizing the quotation process with product configuration', working paper, Technical University of Denmark, 2004.
[2] Huffman, C. and B. E. Kahn, 'Variety for Sale: Mass Customization or Mass Confusion?', *Journal of Retailing* **74** 491–513, 1008.
[3] Hippel, E. v. *Democratizing innovation*, Cambridge, Mass., MIT Press, 2005
[4] Sabin, D. and R. Weigel, 'Product Configuration Frameworks - A Survey', *IEEE Intelligent Systems*, 42–49, 1998.
[5] Salvador, F. and C. Forza, 'Configuring products to address the customization-responsiveness squeeze: A survey of management issues and opportunities', *International journal of production economics*, **91** 273–291, 2004.
[6] Rangaswamy, A. and G. R. Shell, 'Using Computers to Realize Joint Gains in Negotiations: Toward an Electronic Bargaining Table', *Management Science*, **43** 1147–1163, 1997.
[7] Suh, N. P., *The principles of design*, Oxford University Press, New York, 1990.
[8] Raiffa, H., j. Richardson, et al., *Negotiation Analysis: the science and art of collaborative decision making*, Belknap Press, 2003.
[9] Thorsteinsson, E. S. and G. Ottosson, 'Linear Relaxations and Reduced-Cost Based Propagation of Continuous Variable Subscripts.' *Annals of Operations Research* **115** 15–29, 2002
[10] Schniederjans, M.J., *Goal programming: methodology and applications*, Boston, Kluwer Academic Publishers. 1995.
[11] Spronk, J., *Interactive multiple goal programming: applications to financial planning*, Boston, M. Nijhoff., 1981.

Comparing Different Logic-Based Representations of Automotive Parts Lists

Carsten Sinz [1]

Abstract. Parts lists in the automotive industry can be of considerable size. For the Mercedes cars of DaimlerChrysler, for example, they consist of more than 30.000 entries for the larger model lines. Selection of the right parts for a particular product instance is complicated, and typically done via a logical formalism relating order codes with parts. To simplify part assignment, formalisms which use compact and concise formulae are required. We present and compare five different formalisms for such compact logical representations.

1 INTRODUCTION

In the automotive industry there is a persistent trend towards individually configured cars [1, 9]. This results in an enormous product variety that has to be coped with in sales, engineering, production, and after sales. Typically, the configuration of an individual car is accomplished on the level of *order codes*, which represent equipment options that a customer can select [3]. Such options include, among others, different engine types, wheel designs, interior and exterior colors, as well as car electronics like audio and navigation systems, and accessories like bike or ski carriers. As different equipment options may be mutually exclusive or require additional options, formalisms are needed to describe valid combinations. Moreover, automatic checking algorithms are needed to verify whether a customer's order is valid. A common way to describe these constrains is via logical formulae or rules [4, 5, 6, 7, 8, 10].

For each valid order (which satisfies all configuration constraints), in a second step, the right parts have to be selected. Mathematically speaking, this requires a mapping $M : \mathbb{P}(\mathcal{C}) \to \mathbb{P}(\mathcal{P})$ from sets of order codes to sets of parts (we denote by $\mathcal{C}$ the set of order codes, by $\mathcal{P}$ the set of parts, and by $\mathbb{P}(X)$ the powerset of X). Typically, the mapping can be broken down into a sequence of smaller mappings $M_1, \ldots, M_k$, one for each assembly position in the car. For an order $S \subseteq \mathcal{C}$, the required parts list $M(S)$ then is the collection of the required parts for all assembly positions, i.e. $M(S) = M_1(S) \cup \cdots \cup M_k(S)$. Moreover, the mapping for a position is often functional (but not necessarily total), such that the definition of M_i can be changed to $M_i : \mathbb{P}(\mathcal{C}) \to P$, and $M(S)$ becomes $\{M_i(S) \mid 1 \leq i \leq k\}$. We will assume such functional mappings in the rest of this paper.

There are different ways to represent these mappings, and choosing a suitable one is a non-trivial task, as it has to be concise, intelligible, as well as easily maintainable. In what follows, we restrict our attention to parts mappings for individual assembly positions, i.e. we are only interested in constructing the smaller mappings M_i. This makes a difference only from a practical point of view (sizes of considered parts sets), and has no influence on the proposed mathematical formalisms. It should also be noted that the mappings M_i not only have a reduced range (of zero or one in the functional case), but also typically depend only on a few dozen of codes, and thus can also be considered to possess a reduced domain.

[1] Johannes Kepler University, Linz, Austria, email: carsten.sinz@jku.at

2 PARTS LIST REPRESENTATIONS

We now turn to the question, how such parts list mappings M_i can be represented. Throughout this section, we use the following small example to illustrate the proposed methods: Assume three different order codes A, B, and C that influence an assembly position, and four different parts P1, ..., P4 which may be selected depending on the combination of selected codes (in reality there are up to a few dozen of codes and comparably many parts that have to be considered for each position[2]). In each valid configuration we assume that at least one of the codes A, B, C has to be present, and if A and B are selected, then C must also be present in the order. We further assume a parts mapping according to the following variant table:

variant	A	B	C	part
1	X			P1
2		X		P2
3			X	–
4	X		X	P3
5		X	X	P2
6	X	X	X	P4

Note that direct use of such a table is not feasible in practice, as, e.g., for a position depending on 20 codes it would contain up to 2^{20} lines.

2.1 Direct Propositional Encoding

The direct propositional encoding of the parts map uses propositional logic formulae (*parts rules*) that are associated with each part of a position. The parts rule is built upon the order codes, which are used as atomic propositions. To determine the matching part for a position, all the position's rules are evaluated based on the assignment induced by the customer's order (its characteristic function), and those parts for which the rule evaluates to true are selected. In our example, we would therefore obtain the following rule table:

parts rule	part
$A \wedge \neg B \wedge \neg C$	P1
$\neg A \wedge B$	P2
$A \wedge \neg B \wedge C$	P3
$A \wedge B \wedge C$	P4

[2] The largest position for Mercedes' E-Class limousines depends on 135 order codes and contains 27 different parts.

For the order $\{A, C\}$, e.g., rule $A \wedge \neg B \wedge C$ evaluates to true, and thus part P3 is selected.

However, this representation suffers from the drawback that negated codes have to be mentioned, too, which can cause a blow-up of the rules and make them harder to construct and maintain. It thus would be preferable to have a formalism that allows for more compact rules.

2.2 Propositional Encoding with Implicit Negations

The propositional encoding with implicit negations (IN) avoids specification of negated codes in rules and thus delivers a more compact encoding. Rules are computed from the variant table by building a *term* (conjunction of literals) for each row, removing negated codes from each term, and disjunctively composing terms that correspond to the same part. The resulting table is as follows:

IN parts rule	part
A	P1
$B \vee (B \wedge C)$	P2
$A \wedge C$	P3
$A \wedge B \wedge C$	P4

Now when computing the parts assignment for a particular order, an additional decoding step is required before evaluating the rules with the ordinary propositional semantics. This decoding works in two steps, inverting the encoding process:

1. First, all rules are converted to disjunctive normal form (DNF), such that they become disjunctions of terms (conjunctions).
2. Then, for each term, missing codes are added negatedly. Missing codes are codes that occur in the position, but not in the term.

After decoding, rules are evaluated as usual and the suitable part is computed as with the direct propositional encoding.

For the rule of part P2 (which is already in DNF), e.g., decoding delivers $(B \wedge \neg A \wedge \neg C) \vee (B \wedge C \wedge \neg A)$, which is equivalent to $B \wedge \neg A$ (the same as in the direct encoding). Care has to be taken in formulating the shortened IN rules in this formalism, however, as the absorption rule of Boolean logic is not valid any more. Thus, the rule for part P2 must not be simplified to B. Shortened rules of this formalism cannot only be derived from the variant table, but are supposed to be set up straightaway by the parts list maintenance personnel. Note, however, that the IN formalism requires one term for each row of the variant table for which a part is selected, which puts a natural limit on the compression capabilities of this formalism.

2.3 Propositional Encoding with Implicit Exclusions

A slight variant of the propositional encoding with implicit negations is that with implicit exclusion (IE). Like the former, it adds negated subformulae to terms of shortened rules in DNF and requires a decoding step to interpret rules; but in contrast to the former, it does not add missing literals, but rules of other parts, so-called exclusion formulae. The idea of exclusion formulae is to disambiguate part selection for overlapping rules by not assigning any part to the overlap. In more detail, the decoding step works as follows:

1. Compute the DNF of all rules, resulting in a set of conjunctions (terms) for each rule.
2. For each term T of each rule, conjunctively add negations of all terms S from other rules that are not subsumed by T, i.e. for which $S \not\subseteq T$ holds (S and T are regarded as sets of literals here).

As an example, consider the following table with IE rules:

IE parts rule	part
A	P1
$B \vee (B \wedge C)$	P2
$A \wedge C$	P3
$A \wedge B \wedge C$	P4

To decode the first IE rule (for part P1) we have to add negatedly all non-subsumed terms from rules of other parts. These non-subsumed exclusion terms are B, $B \wedge C$, $A \wedge C$ and $A \wedge B \wedge C$, such that the decoded rule for part P1 becomes $A \wedge \neg B \wedge \neg(B \wedge C) \wedge \neg(A \wedge C) \wedge \neg(A \wedge B \wedge C)$, which is logically equivalent to $A \wedge \neg B \wedge \neg C$. As a result, all overlaps with other parts are removed.

2.4 Propositional Encoding with Rule Priority

Another way to achieve more compact rules is by assigning them an evaluation order. This can be done by adding priorities (RP). Rules are then evaluated in order of decreasing priority. As soon as a rule matches, the decoding process is aborted and the respective part is selected. Using priorities we obtain a table like this for our example:

RP parts rule	priority	part
$A \wedge B \wedge C$	3	P4
B	2	P2
$A \wedge C$	2	P3
A	1	P1

Now, for a customer's order $\{A, C\}$, the rules are checked one by one in order of decreasing priority, starting with the rule for part P4, which has highest priority. As this rule does not match, we proceed to any rule with next highest priority (2 in our case), from which the one for part P3 matches. So this part is selected, and the decoding process is finished. A general rule of thumb to assign priorities—to rules consisting of only one term, at least—is to use the number of literals in the term. The RP formalism is used, e.g., in SAP Automotive.

2.5 Cascaded Conditions Algorithm

If priorities assigned in the RP formalism are all distinct, the formalism can be re-written in a more programmatic way, as it then is equivalent to a cascade (CC) of *if-then-else* expressions (or, alternatively, a *case* statement). Modifying the priorities in turn to 4, 2, 3 and 1 for the rows of our exemplary RP table, we obtain this program:

if $A \wedge B \wedge C$ **then** select(P4)
else if $A \wedge C$ **then** select(P3)
else if B **then** select(P2)
else if A **then** select(P1)

One problem with the *if-then-else*-cascades is that they are hard to maintain, especially if the rules and number of cases grow larger. Imagine, e.g., what would happen if $\{A, B\}$ became a valid order that selects no part? Which rules have to be changed in which way then? Maintenance can thus become a non-trivial task.

2.6 Best-Fit Algorithm

At last, we want to present an algorithm that avoids ordering of parts rules for evaluation, but still keeps advantageous properties of the priority-based approach and combines them with ideas from the IE formalism. It works by computing the quality of how good a rule fits

to a customer's order, and selects the best fitting one (BF). Different fitness (or quality) measures are possible, but we only present one that maximizes the number of matching literals. It works as follows:

1. Compute DNFs for all rules, giving a set of terms for each rule.
2. For each term that matches the order (i.e. evaluates to true), compute the number of literals that coincide with the order (matching positive literals that occur in the order as well as negative literals not occuring in the order are counted); this is the fitness measure.
3. If there is exactly one matching term with highest fitness measure, the corresponding part is included into the parts list. Otherwise (i.e. if no or more than one term with highest fitness measure matches), an ambiguity exists, and no part is chosen.

Consider again our examplary table, now with BF parts rules:

BF parts rule	part
A	P1
$B \vee (B \wedge C)$	P2
$A \wedge C$	P3
$A \wedge B \wedge C$	P4

For the order $\{A, C\}$ there are two matching terms, namely A and $A \wedge C$. The latter's fitness measure is 2, whereas the former's is 1. Thus part P3, corresponding to parts rule $A \wedge C$, is chosen.

3 COMPARISON

We now want to compare the aforementioned formalisms, starting with the IN and IE encodings. They seem quite similar, but differences become discernible on even small examples. Consider two parts rules, A and $B \wedge C$:

IN / IE parts rule	IN decoding	IE decoding	part
A	$A \wedge \neg B \wedge \neg C$	$A \wedge (\neg B \vee \neg C)$	P1
$B \wedge C$	$\neg A \wedge B \wedge C$	$\neg A \wedge B \wedge C$	P2

If we visualize the IN / IE part rules in a Venn diagram (Fig. 1 left, red/darker for the first, green/lighter for the second parts rule), we see that there is an overlap between the rules for an order containing all three codes A, B and C. The IN encoding (Fig. 1 middle) selects part P1 only if none of the codes B and C is present, whereas the IE decoding (Fig. 1 right) selects a part for all but the overlap, which is perhaps the more natural interpretation.

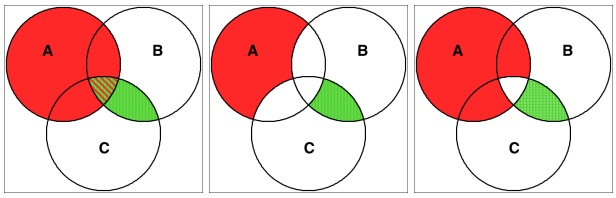


Figure 1. Different interpretations of IN and IE encoding. Left: IN / IE rule; middle: IN decoding; right: IE decoding.

Turning our attention now to all five logical parts list representations, we want to compare them regarding compactness of representation, intelligibility and ease of maintenance.

property	IN	IE	RP	CC	BF
compactness	–	+	+	+	+
intelligibility	0	0	0	+	0
ease of maintenance	–	–	0	–	0

In the encoding with implicit negation (IN) compactness suffers due to the fact that it requires one term for each entry of the variant table. This is not the case for all other encodings, which thus are more concise. Turning to intelligibility, all encodings should be comprehensible after some practice. However, the program-like encoding CC is perhaps the easiest to grasp. Maintaining a (large) logic-based parts list is not simple. This is especially the case for the IN encoding with its resulting bulky rules, but also for the IE encoding, where it might become hard for large rules to figure out the occurring overlaps. The same holds for the CC encoding, where many rules may have to be modified when inserting a new variant.

4 RELATED WORK

In knowledge representation similar problems like those of this paper arise. The most frequently proposed solution is that of assigning priorities (or weights) to rules [11]. Other related formalisms are negation-as-failure (cf. Prolog) or the stable model semantics [2].

5 CONCLUSION

We have presented five different ways to compactly represent logic-based parts lists. The relevance of these formalisms stems from the fact that they are already in practical use at different automotive companies. However, maintenance of logic-based parts lists is a complicated task that requires a thorough understanding of the basic logical formalism. Most of the methods presented in this paper become much more apprehensible when they are accompanied by tool support. In the IE or BF formalisms, e.g., a tool that shows rule overlaps would be very helpful.

In general, we take up the position that rule compilation and maintenance should be considered a programming task. As such, it could benefit from established software engineering methods like coding style-guides, testing or verification.

REFERENCES

[1] S. M. Davis, *Future Perfect*, Addison-Wesley, 1987.
[2] M. Gelfond and V. Lifschitz, 'The stable model semantics for logic programming', in *Proc. 5th Intl. Conf. on Logic Programming*, pp. 1070–1080. The MIT Press, (1988).
[3] A. Haag, 'Sales configuration in business processes', *IEEE Intelligent Systems*, **13**(4), 78–85, (July/August 1998).
[4] D. Mailharro, 'A classification and constraint-based framework for configuration', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)*, **12**(4), 383–397, (1998).
[5] D.L. McGuinness, 'Configuration', in *The Description Logic Handbook*, eds., F. Baader, D. McGuinness, P. Nardi, and P. Patel-Schneider, 397–414, Cambridge University Press, (2003).
[6] S. Mittal and F. Frayman, 'Towards a generic model of configuration tasks', in *Proc. of the 11th Intl. Joint Conf. on Artificial Intelligence*, pp. 1395–1401, Detroit, MI, (August 1989).
[7] D. Sabin and E.C. Freuder, 'Configuration as composite constraint satisfaction', in *Proc. Artificial Intelligence and Manufacturing Research Planning Workshop*, ed., G.F. Luger, pp. 153–161, Albuquerque, NM, (1996). AAAI Press.
[8] D. Sabin and R. Weigel, 'Product configuration frameworks – a survey', *IEEE Intelligent Systems*, **13**(4), 42–49, (July/August 1998).
[9] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin, 'Formal methods for the validation of automotive product configuration data', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)*, **17**(1), 75–97, (January 2003).
[10] M. Stumptner, 'An overview of knowledge-based configuration', *AI Communications*, **10**(2), 111–125, (1997).
[11] R.J. Waldinger and M.E. Stickel, 'Proving properties of rule based systems', *Intl. J. Software Engineering and Knowledge Engineering*, **2**(1), 121–144, (1992).

HP Rack Placement Optimization Case Study

Daniel Naus [1]

Abstract. Hewlett-Packard designs and manufactures high-end computer datacenter solutions, including a wide range of computing, data storage and communication devices. Assembling such solutions requires compliance with a number of constraints, including physical (heat dissipation, electrical), safety, ergonomic and regulatory, as well as optimization of solution cost, high-availability, scalability and serviceability.

This paper presents a case study of the datacenter solution configuration problem, focusing on assisted physical device placement in an e-commerce scenario and illustrates the challenges of implementing both constraint satisfaction and optimization techniques using SAP Internet Pricing and Configurator.

1 Rack placement problem

1.1 Vocabulary

Let us first define the vocabulary used to define the requirements. Computer datacenters are composed of connected isles of cabinets, called racks. Each rack houses a number of computer devices, called rackables.

Rackables can be placed at discrete locations within the rack, this process is called racking. Each rack itself also has a varying number of components, such as doors, panels and mounting kits called rack material. Racks can be tied to physically adjacent racks on the left and the right.

The large majority of rackables have configurable subcomponents, such as hard disk drives, processors, and power supplies, however the description of these and their constraints is not the subject of this paper. Each rackable will be considered as fully configured and described in terms of aggregate requirements on its environment, such as total power, heat dissipation, weight, etc.

1.2 Constraints

Apart from the obvious constraint of all rackables fitting physically within their respective racks, there are a number of less obvious constraints, such as:

- **Electric:** each device must be properly powered within its operating range, taking into account peak loads such as initial power-up. Also, some data connections suffer from decreasing signal strength and must therefore be limited in length.
- **Heat dissipation:** each rackable dissipates heat which must be properly evacuated from the rack. Satisfying this constraint often requires additions of rack fans or, in extreme cases, re-racking to distribute heat dissipating rackables across racks.
- **Ergonomic:** keyboards, monitors and physical switches can only be placed in predefined areas, to comply with local ergonomic requirements.
- **Safety:** majority of rackables is equipped with rails and can slide in and out of the rack, which forces a complex set of stability requirements. In some cases, this requires rack center of gravity computations and additions of weight at the bottom of the rack (called ballast), to ensure the rack will not topple over if a heavy device is pulled out on its rails.

1.3 Optimization requirements

Satisfying the constraints above is a given, however most of the value added of an assisted configuration process is in optimizing the following parameters of the solution:

- **High-availability:** rackables providing power such as uninterruptible power supplies and rack materials for power distribution can be configured to ensure redundant power paths to eliminate single point of failure in the power system. Similar redundancy requirements apply to networking and computer-storage data connections.
- **Scalability:** datacenter solutions have a lifespan of years and sometimes decades and the ability of the system to handle additional load is often expressed by multiplying peak current parameters by a scalability multiplier (larger than 1), which can vary for different rackables (e.g. lower power buffer for expansion, however higher data storage capacity buffer).
- **Serviceability:** all rackables must be easily accessible and to the extent possible, connectivity between devices should be confined to a single rack. This facilitates upgrades, maintenance and can result in significant time savings in case of device malfunction.
- **Cost:** final and often the most important requirement is to satisfy all constrains and optimize all functional requirements with the lowest possible cost. This does not always translate into minimizing unused fractional resources (power, free space in racks), as sometimes a solution with a larger number of less-than-fully utilized cheaper components can be more cost efficient.

2 User interaction requirements

Most previous implementations of assisted datacenter configuration were focusing on the technical sales representative or a pre-sales technical expert, both very familiar with the products and capable of resolving often conflicting optimization requirements. In order to address increasing competition in the high-tech market and cut sales costs, a novel approach is identified in this paper addressing directly the end users of the datacenter solution and forgoing some of the flexibility advanced users require.

[1] Hewlett-Packard, U.S.A, email: daniel.naus@hp.com

End users are not as familiar with the detailed operational parameters of rackables and some of the constraints on racking and do not always require detailed ability to control placement or to modify the configuration result. Often, they have a clear set of high-level preferences and parameters that can be used to drive an automated optimization process. Such preferences can be expressed as:

- What is the minimum time for rackables in a given rack or the whole solution to remain operational in case of a power shortage?
- Should there be an additional safety buffer to account for powering devices added after installation?
- Do you require a fully redundant power infrastructure without a single point of failure?
- Do you prefer a vertically optimized solution (through taller racks and vertical rack extensions) or a standard rack cabinet size for all racks?
- Will you control your rackables using a remote network access or will you require a monitor and a keyboard to be placed in racks? If the latter, how many racks should a given monitoring location control (one, two or more)?

These high-level preferences are to be used as input to a fully automated configuration and optimization process which needs to propose a single solution respecting all of the constraints above and attempt to optimize the overall cost of the solution.

The presentation of the proposed solution must be made in a graphical format, showing in detail all the rackables, their placement and connectivity, which can also be used as assembly instructions at the customer site, in case the whole solution can not be shipped integrated:

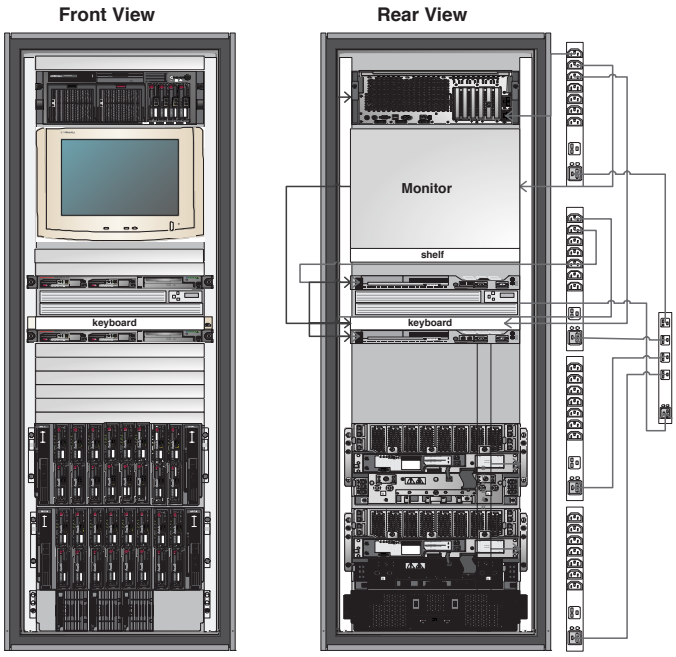


Figure 1. Example rack diagram

3 Design & Implementation

3.1 Design Decisions

Several decisions were taken early on in the implementation of the configuration solution in order to leverage existing data sources at HP and speed up implementation time:

- Reuse of SAP configuration models of individual rackables.
- Use of Visio as the graphical layout platform and leverage of existing stencils of HP products.
- Use of SAP Internet Pricing and Configurator (IPC) as the platform maintaining the solution configuration content and enforcing all declarative constraints.
- Implementation of optimization logic in custom Java code accessing SAP IPC APIs.
- User interface leveraging the standard SAP IPC JSP UI.
- Reuse of existing Excel-based calculation spreadsheets for detailed component calculations.

3.2 IPC Model

The configuration model had to be capable of expressing all the necessary solution attributes and relationships. SAP provides the following configuration products:

- SAP R/3 Configurator (called Variant configurator), which is used within the mySAP ERP system.
- SAP Internet Pricing and Configurator (IPC), which is currently used in mySAP CRM and internet sales scenarios.

Variant configurator only executes within the mySAP ERP system and can not be run in a standalone mode, so it was rejected as the configuration execution environment for our project.

SAP IPC is a Java-based application, which can run standalone and allows execution of configuration models that are either compatible with the Variant Configurator (with all of its limitations), or take advantage of additional SAP IPC capabilities (so called advanced models).

Given our requirements, the IPC Advanced mode modeling was chosen, as it allows for arbitrary relationships, uses a more flexible decomposition structure and has additional summation and aggregation capabilities.

The core model in IPC is composed of three major components: classification hierarchy, variant tables and constraints. Classification hierarchy contains classes, their attributes called characteristics and class to class links. Variant tables are used to store characteristic values for products allocated to classes and also used as support for constraints in cases where constrained relationships are best represented as tabular combinations of characteristic values. Constraints are used to express all of the declarative constraint requirements above.

Classification hierarchy design was governed by a small set of best practices as identified by previous experience with SAP IPC:

1. Use a minimal and shallow class hierarchy capable of supporting all of the constraints required. In other words, do not create new classes unless there is at least one constraint that requires them.
2. Avoid multiple inheritance.
3. Build the hierarchy from the bottom up, incrementally allowing characteristics to move up the hierarchy if a generic rule requires them at higher levels.

Following these guidelines resulted in a relatively small class hierarchy of about 150 classes with up to 3 levels of inheritance. About 40 different relationships (such as connected to, adjacent to, pulling power from, etc.) were defined as part of the class hierarchy.

Variant tables are convenient way of expressing legal combinations of features in a tabular form and were employed to store static attributes of rackables and racks, such as dimensions, physical locations of ports, and operating limits. Tables were also used to express tabular relationships, such as compatible port to connector types. Variant tables are not used in quite the same way as relational database tables, as often sets of records from a table are retrieved at once in support of imposing restrictions and often no single primary/foreign keys are defined as there are multiple constraints, each using the table in a different manner. Not all best practices for database design can therefore be transported without modification, but the fundamental ones do apply and were followed:

1. Clearly separate entity tables from relationship tables.
2. Map out and maintain foreign key relationships.
3. Attempt to normalize tables and eliminate white space. This rule is broken at times as it conflicts with table ease-of-use maintenance requirements.

This resulted in about 100 variant tables, from 2 up to 15 columns wide, each with typically up to several hundred records.

Finally, the constraint design followed as well a small number of practical rules of thumb in an effort to maximize consistency and maintainability:

1. Group constraints into constraint nets by the principal class nodes that they apply to. Use consistent naming convention shared between the class node names and its corresponding constraint net and its constraints.
2. Apply constraints as high up in the class hierarchy as possible and so eliminate redundant constraints in parallel branches.
3. Keep constraints atomic, performing as few inferences/restrictions as possible, in order to facilitate changes in their class allocation.

SAP unfortunately does not support moving constraints from one constraint net to another, so constraints were often cut and pasted from constraint nets at lower class hierarchy levels to nets higher up.

This constraint design resulted in less than 2000 constraints, including those used in internal rackable subcomponent configurations. Each constraint consists typically of 10 to 20 lines of constraint source code. As an example, a typical rule in datacenter configurations is that all devices within the same cabinet have to support the same voltage. The constraint syntax for this requirement is as follows:

```
Objects:
    ?Rackable is_a (300) HP_Rackable
    WHERE
       ?Rack = is_racked_in
Restrictions:
    ?Rackable.Voltage = ?Rack.Voltage
Inferences:
    ?Rack.Voltage, ?Rackable.Voltage
```

3.3 Racking algorithm

SAP IPC advanced mode does not support any optimization capabilities such as solution space search including backtracking and procedural reasoning (for sorting and algorithmic logic), so the only available design was to create the rack optimization logic in Java (called the racking algorithm). This java code had to have the following fundamental features:

1. Ability to run incrementally, after a change in user preferences or addition/deletion of rackables.
2. Use only configuration information contained within the IPC configuration model.
3. Minimize unproductive constraint engine execution when exploring different optimization options (called churning).
4. Execute as fast as possible.

Development of the racking algorithm proved to be one of the biggest challenges of the project. SAP IPC was not designed to be used with an external optimization process and it turned out to be rather expensive to maintain the complete configuration state there. Also, the racking algorithm had to correctly recognize and detect internal state changes of the model in IPC.

At a high-level, the racking algorithm performs the following steps:

1. Disassociate any currently placed rackables from their respective racks.
2. Create as many racks as are necessary for physically housing all of the rackables.
3. Group closely connected devices into groups (e.g. server and storage devices connected together, ideally such groups can be directly identified by users).
4. Sort devices within a group by weight and start placement at the bottom of the first rack.
5. Place all devices in racks while respecting all of their published restrictions (such as allowed locations, adjacency requirements, etc.)
6. Constantly check the rack model state as devices are being placed and backtrack if any inconsistencies are found and attempt a different placement.

The following learnings can be inferred from our experience:

- Model steps in the optimization process as discrete states within the IPC configuration model and only trigger constraints appropriate for any given step. At minimum, three different states are needed: prior to racking, racking execution, and post-racking state.
- Try to perform as much constraint driven logic in the pre-racking phase and publish resulting restrictions through allowed characteristic values.
- Clearly identify which constraints in the racking phase can trigger an inconsistency (e.g. rack stability constraint can be violated as devices are being placed). Keep these constraints as simple as possible.
- Where applicable, define constraints that can be used to work around inconsistencies, for example solving a rack stability problem by instantiating additional ballast.

Even with these guidelines in mind, execution of the racking algorithm turned out to be rather expensive (from 10 seconds to several minutes for very large configurations).

3.4 External calculation

Implementing some of the calculations necessary in high-end rackable products (such as precise power calculation) would not be practical using constraint technology due to the modeling effort required

and the overhead incurred by keeping track of extremely detailed power calculation values as facts in the constraint engine. Instead, existing Excel-based calculation spreadsheets were identified, mapped using a rigorous mapping to IPC model characteristic values and then translated into native java classes using an off-the-shelf Java-based spreadsheet engine from Actuate.

Our experience with this design has been very positive and very large and complex engineering calculation spreadsheets with thousands of formulas were leveraged, further optimizing the cost of the solution (as opposed to using aggregate estimated requirements).

3.5 User interaction

At a high-level, configuration session consists of the following steps:

1. Specify/modify high-level configuration preferences.
2. Instantiate all of the rackables required and configure them fully, including defining groups of rackables (such as servers and storage connected together).
3. Execute the racking algorithm.
4. Review the results and if dissatisfied, repeat the process incrementally applying changes in steps 1-2 and re-execute 3.

Thanks to largely automating the racking logic inside of the racking algorithm, the user entry screens were mostly simple pages with drop down menus and checkboxes. A tree-control was used to navigate the configuration content before and after the racking execution.

Our experience with the standard SAP Java Server Pages (JSP) Configuration Interface was positive overall, even though the intended use of it was only for demonstrations and testing. Several areas of improvement were identified and customized:

- Configuration content navigation had to be significantly improved to navigate by relationship rather than by bill-of-material decomposition.
- Custom report screens and output capabilities were added (e.g. for configuration result, power summary, saving of the configuration or its graphical output, etc.)

Visualization of the configuration solution was built using the Microsoft Visio diagramming platform, as SAP does not provide a comparable visualization product. As we did not have a requirement for an interactive graphical display, we decided on the following simplifying design choices:

- Separate drawing logic completely from the IPC model execution logic.
- Create a static mapping of Visio stencils to rackables and racks, identifying connection points for different connectivity ports.
- Perform an export of all relevant drawing attributes (such as location, connections, etc.) at the time of digram generation.
- Generate a Microsoft script for drawing of the whole diagram, used to control the Visio drawing engine.
- Display the resulting Visio diagram in a web-based viewer.

The quality of the resulting output proved to be excellent, thanks to outstanding diagramming features of Visio, however the overhead associated with the diagram generation also proved to be rather large. Also, this solution can not be recommended if a dynamic interactive diagram is required, unless the visualization architecture would be enhanced to handle incremental changes. Building of the visualization layer was a significant investment (6 man months).

4 Conclusion

The results of this thorough study demonstrate that an assisted configuration of computer datacenter solutions requires a technology that natively supports both declarative constraint-based reasoning as well as optimization capabilities (solution search and backtracking).

Our study did confirm feasibility of custom integration between a purely declarative engine and a custom optimization process, however the difficulties encountered, performance characteristics and the complexity of the resulting custom code warrant a native solution integrated within the configuration engine.

As part of our study, we have identified a number of simple modeling guidelines, which when applied consistently, lead to a rather small footprint of the constraint model required to support datacenter configuration.

It is also highly desirable to support callouts to external calculation engines, such as java-based spreadsheet engines, as this simplifies the core constraint design.

Visualization using Microsoft Visio is only recommended for non-interactive scenarios requiring either exceptional drawing quality or the ability to perform further modifications such as markups on the resulting diagram.

Integrating Knowledge-Based Product Configuration and Product Line Engineering: An Industrial Example

Rick Rabiser and **Deepak Dhungana** and **Paul Grünbacher**[1]

Abstract. The software product line and product configuration research communities have evolved quite independently in the last couple of years. More recently, however, researchers have started working on the integration of software product line engineering and (knowledge-based) product configuration approaches. In this paper we describe an industrial example that suggests closer integration of the two research fields. We propose a tentative approach that uses concepts from both areas.

1 MOTIVATION

A software product line has been defined as "a set of software-intensive systems sharing a common, managed set of features that satisfy the needs of a particular market segment and that are developed from a common set of core assets in a prescribed way" [4]. Product line engineering (PLE) is the discipline of creating and managing software product lines. It aims at reducing cost and increasing productivity and reliability through leveraging reuse of artefacts and processes in particular domains [15].

Product configuration is concerned with automatic configuration of technical products based on product models and well-defined rules. Product configurators are among the most successful applications of artificial intelligence (AI) technology and have been successfully applied in industrial environments [3, 8, 9]. The major contribution of configuration systems is to enable the configuration of complex products and services in shorter time and with less errors. It has been demonstrated that by using configurators, individually customized products can be configured at much lower costs.

A combination of the two approaches seems reasonable because configuration approaches complement certain important activities in PLE. Core assets of a product line (including their variabilities and commonalities) are typically described formally using languages and models such as feature models [2], architectural models [5], or decision models [16]. When deriving individual products, these assets need to be customized and configured. The identification of valid combinations of these assets and the automatic verification of the correctness and completeness of individual products can be supported with existing configuration techniques. Various other AI techniques (used in configuration approaches) like SAT Solving [14] can support PLE to deal with the complexity of real-world systems. By integrating PLE with product configuration we expect to improve our industrial partner's overall product derivation process. Many activities in this process are carried out by non-experts (the sales staff), who could also benefit from the mentioned integration.

[1] Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University, A-4040 Linz, rabiser@ase.jku.at.

The remainder of this paper is structured as follows: In Section 2 we describe challenges our industry partner in product customization. Section 3 describes our tentative approach for integrating knowledge-based product configuration with PLE to overcome the challenges discussed in Section 2. In Section 4 we briefly describe related work. Section 5 rounds out the paper with a conclusion and some issues for discussion.

2 INDUSTRIAL EXAMPLE

Our industry partner Siemens VAI is the world's leading engineering and plant-building company for the iron, steel, and aluminum industries. In our ongoing research cooperation we apply a product-line approach to automatically generate software from feature specifications that characterize individual members of the product line. One important objective is the creation of an innovative approach for making feature-based product configuration accessible to non-software-experts. The expected outcomes of integrating software PLE and product configuration approaches are an improved sales process and an accelerated, less error-prone product configuration process.

The software system developed by Siemens VAI automates continuous casting in steel plants and provides capabilities for process supervision, material tracking, and process optimization. The size is about 1.3 million lines of Java code. The software has a state-of-the-art component-oriented architecture which makes it highly configurable, customizable, and extendable. Each year our industry partner delivers about 20-30 software solutions customized to the particular needs of individual customers. The configuration and customization work is accomplished by around 40 software engineers only. Due to the size and complexity of the software there are many non-trivial dependencies among the available features as well as between features, architectural elements, and technical solution components.

Currently there exists a gap between the developers and the sales people of our industrial partner, a well-known problem of many companies. Sales people do not have access to architectural knowledge needed to customize the product on feature-level. At the same time developers are not informed adequately about new customer requirements and special wishes. The sales staff use Excel spreadsheets to communicate the features of the software to the customers. The selected features are then used by the software engineers to derive a specific product from the product line. Several problems arise from this manual form of configuration process. For instance, (1) the feature lists are often incomplete or not up to date and (2) the lists do not take into account the dependencies between features and the underlying technical solution. The overall goal of the research project is to provide better guidance for sales people and improve the communication between sales staff and engineers. In this way, our partner

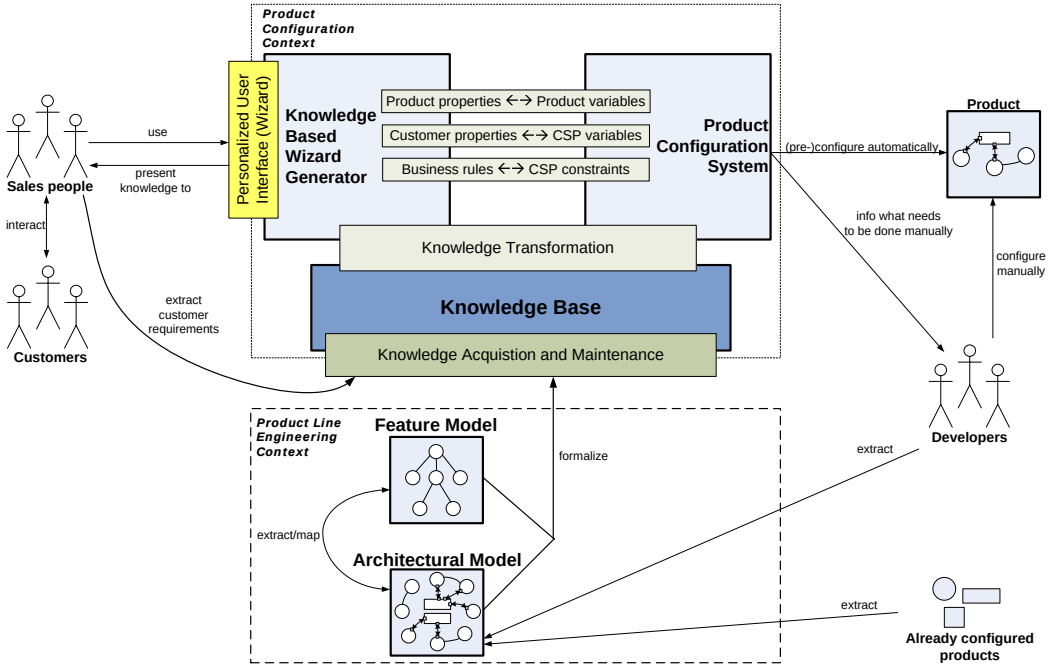


Figure 1. Knowledge-based approach integrating product line engineering and product configuration concepts (product configuration context based on [6])

expects a considerable productivity boost and significant reduction of economic risks. The customers benefit from a shorter time-to-market and products better customized and configured to their specific needs.

Apparently, an automated configuration and verification approach during feature selection and product derivation is missing in the current scenario. The absence of such an approach increases the likelihood for errors in the product configuration process. It is therefore necessary to ascertain that the features selected by the customers are consistent with the underlying technical solution and that the combination of the selected features is technically feasible. In the next section, we explain how we expect a knowledge-based configuration approach to address these problems.

3 TENTATIVE APPROACH

Our proposed approach relies on the existence of a product line variability model (PLE context, see Figure 1). Such a model covers the features [2] and architecture [5, 13] together with a decision model describing the decisions one needs to take during product configuration [16]. The model thus captures the variability of the product line and explicitly describes the dependencies between different core assets (architectural elements, features, decisions). Due to space limitations the methods for creating these models are not described in this paper. The proposed approach consists of three main steps:

- Generation of a knowledge base from existing models and customer requirements.
- Making the knowledge accessible to non-experts.
- Enabling automatic product configuration.

Generation of a knowledge base from existing models and customer requirements. For building a knowledge base for the purpose of configuration, the information required is gathered from the product line variability model. As illustrated in Figure 1, the knowledge base is created by formalizing product line models (the models in the PLE context, i.e., feature model and architectural model) and customer requirements. It contains information about the core assets (product properties), customer requirements and properties, and business rules (which express the relationships between different assets). Felfernig *et al.* [7] report on an approach to automatically generate such product configuration knowledge bases (see Section 4).

Making the knowledge accessible to non-experts. We consider a tool-supported, interactive presentation of the features as essential to deal with the challenges described in Section 2. To present the features to the customers in a convenient and interactive manner based on their needs is not trivial: On the one hand meta-information about features should be available at all times in different forms (e.g., multimedia feature explanations). On the other hand the dependencies among features and between the features and the underlying technical solution should be constantly verified. Ideally, the consequences of feature selection should be made visible to customers immediately. For example, if they choose a feature that requires other features these should be selected automatically and a proper explanation should be given to the customer.

The knowledge-based wizard generator (Figure 1) is a system for building intelligent, personalized wizards that support the sales advisory process between sales staff and customers. One example for such wizards supporting the sales process are recommender systems [1], successfully applied in e-commerce environments (e.g., [10]). Based on customer properties and wishes concerning the technical solution, the wizard generator calculates which decisions are to be taken and in which order they should be taken. A wizard is generated for each individual customer based on his properties and his specific needs. Such wizards are flexible and interactive which hides the complexity of the underlying system from the customers. The customers select features they need and the sales staff tailor the product to their specific needs. At the same time the underlying constraints and dependencies (business rules) can be checked.

Enabling automatic product configuration. The result of such a presentation of the features is a consistent selection of core assets required for product derivation. Finding a valid configuration of the required assets and verifying its correctness can be supported by a product configurator. Such a tool is based on the same knowledge base as the wizard generator and therefore the concepts used by these tools can be mapped to each other. By and large, a configuration problem can be tracked down to a constraint satisfaction problem (CSP). Product properties can be mapped to product variables, customer properties to CSP variables, and business rules to CSP constraints. A CSP solver [17] can then be used to generate a valid product configuration. Although major parts of the product can be configured automatically with the information received from the sales process, some manual configuration work will still be required. Therefore it is essential that the information about things that need to be done manually is available after configuration. The developers then finish the configuration of the system so that it can be deployed to the customer.

4 RELATED WORK

The product configuration community has been developing approaches to automatically build knowledge bases from existing product models. For example, in [7] the authors propose an approach for generating a valid configuration knowledge base based on a UML product model. The scale and complexity of our industry partners software solution is much higher than those of the examples (i.e. a configurable PC) described in [7] and other papers focussing on knowledge-based configuration problems [9, 11]. Also PLE aspects are not taken into account. However, these papers provide interesting concepts on how to generate a knowledge base from our product line models (feature models, architectural models, decision models).

The Advisor Suite described in [10] is a commercial system for building intelligent, personalized sales advisory applications following a knowledge-based approach. With such applications complex product knowledge is presented to users via a personalized user interface. The users are able to tailor a product to their specific needs while simultaneously underlying business rules are satisfied. Knowledge-based advisor applications provide a candidate solution to present information about our industrial partners complex software product line to the sales people/customers (Figure 1).

Männistö *et al.* [12] compare the areas of software product lines and product configuration of traditional products based on the concepts for modelling variety and evolution. This work is of great interest as the authors also emphasize the integration of product configuration and software product families.

5 CONCLUSION AND OPEN ISSUES

In this paper we described challenges and a candidate solution for integrating concepts from two research communities, the software product line community and the product configuration community. In our tentative approach, product line knowledge is used as a basis for knowledge-based configuration systems. We believe that such a hybrid approach will help us to make feature-based configuration accessible to non-experts. We will provide experiences and prototypes demonstrating the feasibility of our ideas in our further work. Three issues are of particular interest in our research:

Automatic generation of knowledge bases from product line models. As already mentioned in Section 4, approaches exist for automatically generating product configuration knowledge bases (e.g., [7]). We will explore on how to adapt these approaches to fit our software (PLE) context.

Evolution of the software product line. Another interesting question to address is on how to manage evolution of the software product line. A product configuration knowledge base also must evolve in case of changes of the underlying product.

Acceptance of our approach and tools. We will have to keep in mind that our users, i.e., sales people, must be able to use the developed tools. Usability is a prime concern and the approach has to reduce their work load instead of increasing it.

REFERENCES

[1] G. Adomavicius and A. Tuzhilin, 'Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions', *IEEE Transactions on Knowledge and Data Engineering*, **17**(6), 734–749, (2005).

[2] T. Asikainen, T. Männistö, and T. Soininen, 'Representing feature models of software product families', in *16th European Conference on Artificial Intelligence*, (2004).

[3] T. Blecker, N. Abdelkafi, G. Kreuter, and G. Friedrich, 'Product configuration systems: State-of-the-art, conceptualization and extensions', in *Eight Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI)*, pp. 25–36, (2004).

[4] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, SEI Series in Software Engineering, Addison-Wesley, 2005.

[5] E.M. Dashofy, A. van der Hoek, and R.N. Taylor, 'An infrastructure for the rapid development of xml-based architecture description languages', in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pp. 266–276, New York, NY, USA, (2002). ACM Press.

[6] W. Falle, D. Stöfler, C. Russ, M. Zanker, and A. Felfernig, 'Using knowledge-based advisor technology for improved customer satisfaction in the shoe industry', in *International Conference on Economic, Technical and Organisational aspects of Product Configuration Systems*, (2004).

[7] A. Felfernig, G.E. Friedrich, and D. Jannach, 'Uml as domain specific language for the construction of knowledge-based configuration systems', *International Journal of Software Engineering and Knowledge Engineering*, **10**(4), 449–469, (2000).

[8] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner, 'Configuring large systems using generative constraint satisfaction', *IEEE Intelligent Systems*, **13**(4), 59–68, (1998).

[9] A. Günter and C. Kühn, 'Knowledge-based configuration: Survey and future directions', in *XPS '99: Proceedings of the 5th Biannual German Conference on Knowledge-Based Systems*, pp. 47–66, Würzburg, Germany, (1999). Springer.

[10] D. Jannach, 'Advisor suite - a knowledge-based sales advisory-system.', in *16th European Conference on Artificial Intelligence*, pp. 720–724, (2004).

[11] T. Krebs, L. Hotz, and A. Günter, 'Knowledge-based configuration for configuring combined hardware/software systems', in *Proc. of 16. Workshop, Planen, Scheduling und Konfigurieren, Entwerfen (PuK2002)*, Freiburg, Germany, (2002).

[12] T. Männistö, T. Soininen, and R. Sulonen, 'Modelling configurable products and software product families', in *International Joint Conference on Artificial Intelligence (IJCAI)*, (2001).

[13] M. Matinlassi, 'Comparison of software product line architecture design methods: Copa, fast, form, kobra and qada', in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pp. 127–136, Washington, USA, (2004). IEEE Computer Society.

[14] A. Nareyek, E.C. Freuder, R. Fourer, E. Giunchiglia, R.P. Goldman, H.A. Kautz, J. Rintanen, and A. Tate, 'Constraints and ai planning.', *IEEE Intelligent Systems*, **20**(2), 62–72, (2005).

[15] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering : Foundations, Principles and Techniques*, Springer, 2005.

[16] K. Schmid and I. John, 'A customizable approach to full lifecycle variability management', *Sci. Comput. Program.*, **53**(3), 259–284, (2004).

[17] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1995.

How to recommend configurable products?

Alexander Felfernig [1] and **Christian Scheer** and **Peter Loos** [2]

Abstract. Knowledge-based configuration has a long history as a successful application area for AI technologies [2, 15, 10, 8, 5, 9]. Starting with rule-based systems [2], higher level representation formalisms have been developed allowing a more intuitive representation of configurable products. The advantages of these representations are faster application development, higher maintainability, and more flexible reasoning support. These representations have proven their applicability in various real-world applications. What still remains a challenging task for the configuration community is the provision of flexible and intuitive interfaces which alleviate the accessibility of complex product assortments for customers. A first step towards this direction has been conducted within the scope of the CAWICOMS project [1], where multi-attribute object rating approaches and rule-based adaptation technologies have been developed in order to improve the accessibility of configurable products. The goal of our work is to provide a general overview of recommendation approaches [3, 4, 6, 7, 11, 12, 13, 14, 16, 17] and potential applications in knowledge-based configuration processes (e.g., determination of default values using collaborative filtering approaches [6, 12, 14], etc.).

References

[1] L. Ardissono, A. Felfernig, G. Friedrich, D. Jannach, G. Petrone, R. Schaefer, and M. Zanker, 'A Framework for the development of personalized, distributed web-based configuration systems', *AI Magazine*, **24**(3), 93–108, (2003).

[2] V.E. Barker, D.E. O'Connor, J.D. Bachant, and E. Soloway, 'Expert systems for configuration at Digital: XCON and beyond', *Communications of the ACM*, **32**(3), 298–318, (1989).

[3] R. Burke, 'Knowledge-based Recommender Systems', *Encyclopedia of Library and Information Systems*, **69**(32), (2000).

[4] R. Burke, 'Hybrid Recommender Systems: Survey and Experiments', *User Modeling and User-Adapted Interaction*, **12**(4), 331–370, (2002).

[5] G. Fleischanderl, G. Friedrich, A. Haselboeck, H. Schreiner, and M. Stumptner, 'Configuring Large Systems Using Generative Constraint Satisfaction', *IEEE Intelligent Systems*, **13**(4), 59–68, (1998).

[6] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. Riedl, 'Evaluating Collaborative Filtering Recommender Systems', *ACM Trans. on Information Systems*, **22**(1), 5–53, (2004).

[7] B. Jiang, W. Wang, and I. Benbasat, 'Multimedia-Based Interactive Advising Technology for Online Consumer Decision Support', *Communications of the ACM*, **48**(9), 93–98, (2005).

[8] E.W. Juengst and M. Heinrich, 'Using Resource Balancing to Configure Modular Systems', *IEEE Intelligent Systems*, **13**(4), 50–58, (1998).

[9] D. Mailharro, 'A classification and constraint-based framework for configuration', *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing Journal, Special Issue: Configuration Design*, **12**(4), 383–397, (1998).

[10] S. Mittal and F. Frayman, 'Towards a Generic Model of Configuration Tasks', in *11th International Joint Conference on Artificial Intelligence*, pp. 1395–1401, Detroit, MI, (1990).

[11] M. Pazzani, 'A Framework for Collaborative, Content-Based and Demographic Filtering', *Artificial Intelligence Review*, **13**(5-6), 393–408, (1999).

[12] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl, 'GroupLens: An Open Architecture for Collaborative Filtering of Netnews', in *ACM Conference on Computer Supported Cooperative Work*, pp. 175–186, (1994).

[13] F. Ricci, A. Venturini, D. Cavada, N. Mirzadeh, D. Blaas, and M. Nones, 'Product Recommendation with Interactive Query Management and Twofold Similarity', in *5^{th} International Conference on Case-Based Reasoning (ICCBR 2003)*, pp. 479–493, Trondheim, Norway, (2003).

[14] B. Sarwar, G. Karypis, J. A. Konstan, and J. Riedl, 'Item-based collaborative filtering recommendation algorithms', in *10^{th} International World Wide Web Conference*, pp. 285–295, (2001).

[15] S.Mittal and F. Frayman, 'Towards a Generic Model of Configuration Tasks', in *11^{th} International Joint Conference on Artificial Intelligence*, pp. 1395–1401, Detroit, MI, (1989).

[16] B. Smyth, E. Balfe, O. Boydell, K. Bradley, P. Briggs, M. Coyle, and J. Freyne, 'A Live User Evaluation of Collaborative Web Search', in *19^{th} International Joint Conference on Artificial Intelligence*, pp. 1419–1424, Edinburgh, Scotland, (2005).

[17] C. Thompson, M. Göker, and P. Langley, 'A Personalized System for Conversational Recommendations', *Journal of Artificial Intelligence Research*, **21**, 393–428, (2004).

[1] Institute for Business Informatics and Application Systems, University Klagenfurt, A-9020 Klagenfurt, Austria, email: alexander.felfernig@uni-klu.ac.at

[2] Information Systems and Management, Johannes Gutenberg-Universität Mainz, D-55099 Mainz, Germany, {scheer, loos}@isym.bwl.uni-mainz.de

Extended Abstracts

Knowledge-based composition of recommendations

Extended Abstract

Markus Zanker[1] and **Markus Aschinger**[2] and **Marius Silaghi**[3]

Abstract. Recommender systems help users to orientate themselves when confronted with a large variety of choices. However, in situations where the user desires recommendations on product bundles most systems do not have adequate reasoning capabilities.

In our work we are therefore interested in exploring the problem space of computing composite recommendations. Based on a generic framework architecture we intend to develop a knowledge-based approach that integrates and combines recommendations for different product categories.

1 Motivation

Recommender systems have become commonplace in many online shops over the past decade. They serve online customers by making personalized product proposals that best fit their needs. A variety of different recommendation paradigms has been reported so far: e.g. collaborative and content-based filtering [1, 10] as well as knowledge-based [3, 2, 5] and case-based recommendation [8, 7]. Upon request recommender systems propose one or several items out of a large set of product instances to a specific user.

However, in some domains like tourism the recommendation of whole bundles of product instances would be desirable. When planning a trip, for instance recommender systems might not only propose appropriate accommodations to a user but also restaurants s/he would like to go out for dining, sights that might match her/his interest profile or different leisure and sporting activities. This issue of travel planning and recommending has already been addressed by several approaches. Ricci and Werthner [9] propose a case-based travel recommender, that proposes past travel cases based on similar user requirements. Torrens et al. [11] developed a light-weight constraint library for Web applications that can solve travel plans on the client-side, while Knoblock [6] presents a multi-agent planning approach.

However, in our application domain we do not focus on planning a trip schedule that conforms for instance with different flight connections or other stringent time restrictions, but on making different proposals to a tourist during his stay that do not contradict each other.

2 Approach

We will explore a knowledge-based approach, where explicit domain knowledge is used for composing different product recommendations. The problem domain contains a set of different product types T that restrict the product instances that might be recommended like sporting or leisure activities, restaurants or sights. Furthermore, a model of the user U and explicit domain knowledge R is known. The user model describes the user's preferences and needs situation, while R is constituted by a set of restrictions that derive from a domain expert. For instance recommended restaurants must not be located too far away from the hotel the user is staying at or an overall budget restriction must be obeyed.

For modeling and problem solving we are currently evaluating a csp-based approach. A central mediator component requests autonomous recommender systems for ranked lists of product proposals for different product types. Then recommendations are bundled that do not conflict with restrictions in R. For solving we currently employ the Choco constraint library [4]. The following challenges are of interest for our future research:

- Balancing priorities between the different ranked lists of recommendations to create an optimal recommendation bundle,
- intuitive knowledge acquisition for domain experts, and
- distributed solving strategy and dynamic domain extensions when no solutions are found.

[1] University Klagenfurt, Austria, email: markus.zanker@uni-klu.ac.at
[2] University Klagenfurt, Austria, email: masching@edu.uni-klu.ac.at
[3] Florida Institute of Technology, USA, email: msilaghi@fit.edu

REFERENCES

[1] Marko Balabanovic and Yoav Shoham, 'Fab: Content-based, collaborative recommendation', *Communications of the ACM*, **40(3)**, 66–72, (1997).
[2] Robin Burke, 'Knowledge-based recommender systems', *Encyclopedia of Library and Information Systems*, **69**, (2000).
[3] Robin D. Burke, Kristian J. Hammond, and Benjamin C. Young, 'The findme approach to assisted browsing', *IEEE Expert*, **July/Aug.**, 32–40, (1997).
[4] Laburthe Francois, Jussien Narendra, Rochart Guillaume, and Cambazard Hadrien. Choco User Guide.
[5] Dietmar Jannach, 'Advisor suite - a knowledge-based sales advisory system', in *European Conference on Artificial Intelligence - ECAI 2004*, (2004).
[6] Craig Knoblock, 'Building software agents for planning, monitoring, and optimizing travel', in *Proceedings of ENTER*, (2004).
[7] Kevin McCarthy, James Reilly, Lorraine McGinty, and Barry Smyth, 'Experiments in Dynamic Critiquing', in *Proceedings of the Intelligent User Interfaces Conference (IUI)*, pp. 175–182, (2005).
[8] Francesco Ricci and Hannes Werthner, 'Case base querying for travel planning recommendation', *Information Technology and Tourism*, **3**, 215–266, (2002).
[9] Francesco Ricci and Hannes Werthner, 'Case-based querying for travel planning recommendation', *Information Technology and Tourism*, **4(3-4)**, 215–226, (2002).
[10] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl, 'Analysis of recommendation algorithms for e-commerce', in *ACM Conference on e-Commerce (EC)*, pp. 158–167, (2000).
[11] Marc Torrens, Patrick Hertzog, Loic Samson, and Boi Faltings, 'reality: a Scalable 1ntelligent Travel Planner', in *Proceedings of ACM Symposium on Applied Computing (SAC)*, pp. 623–630, (2003).

Industry specific 'Standard Template Libraries' for Configuration

Manikandan Sundaram and Rajasekhar Vinnakota and Praveen Vudoagiri
Trilogy Software Inc.
{manikandan.sundaram | rajasekhar.vinnakota | praveen.vudoagiri}@trilogy.com

Abstract

Developing solutions to real-world high-end configuration problems is extremely complex and effort-intensive. This paper makes a case for reducing the challenges involved by using "Standard Template Libraries" that encapsulate domain-specific solutions and practices. This allows for reuse across configuration deployments of configuration logic patterns that represent the common product characteristics and business processes within the same or similar industries, as well as improving the quality and usability of the deployments by creating specialized run-time and maintenance interfaces that use the industry-specific language familiar to users.

1 INTRODUCTION

Configurators are used to solve complex problems in a wide range of sectors like Automotive, Computer, Telecom, Furniture, Aerospace and Industrial Equipment Manufacturing. Every industry has its own unique set of challenges and uses for the configurator. The solutions to these challenges often can be designed to be generic and reusable within the domain.

For example, racking a chassis such that rack space is optimized is a standard challenge encountered in configuration systems for enterprise computer OEMs. There are typical manufacturing considerations like the weight of the chassis, heat dissipation, ease of cabling and safety/ergonomic considerations that impact how the chassis is racked. The solutions to the racking problem based on these considerations are standard across all manufacturers in the enterprise computer space.

Encapsulation of these standard solutions based on the prevalent industry practices, in Standard Template Libraries, aids in quickly deploying robust and scalable configuration models. Each industry will have its own set of libraries that captures the practices and solutions specific to that industry domain.

2 STANDARD TEMPLATE LIBRARY

The Standard Template Library (STL) for a specific industry, in addition to being a collection of highly extensible models, objects and generic configuration logic also is a repository of standard industry practices prevalent in that industry. Access to this repository of accumulated knowledge helps in rapid creation and maintenance of models with complex configuration needs and engineering standards.

A typical problem in the enterprise computer industry is distribution of cards in a optimized fashion. The common practice is to capture generic logic like 'card needs a slot' or 'place case in a cabinet' in reusable functions and methods. However practices like various card-packing techniques for optimization (Bandwidth mode / Performance mode) are not typically addressed. STL for enterprise computer industry will contain algorithms for the standard optimization techniques used for the above problems in the industry. Solution developer can make use of the logic to evolve models that provide optimized solutions to the sales force.

Based on Trilogy's SalesBUILDER configuration products, we have successfully developed and deployed industry-specific frameworks used to create solutions that capture the configuration logic for our Fortune 500 customers in the enterprise computing and telecommunications industries. One key learning in our experience is that it is important to minimize the amount of customization required to represent each customer's product portfolio. By offering multiple alternative implementations to the common industry-standard configuration problems, the solution developer can select the appropriate approach rather than implementing something custom. As proposed, STL is a formalization of this approach used in our solutions.

3 FUTURE WORK

The potential next steps to explore is to enhance the utility of the STL by incorporating the following feature set:

1. Ability to replicate BOM structures from ERP models in to product model.
2. Ability to auto generate user interface for a product model.

4 CONCLUSION

Benefits of STL in configurator will come through in the reduction of effort needed to develop each product. STL provides a streamlined process to evolve a product model in the configurator from requirements. Thereby it reduces the development effort and hence the time to market.

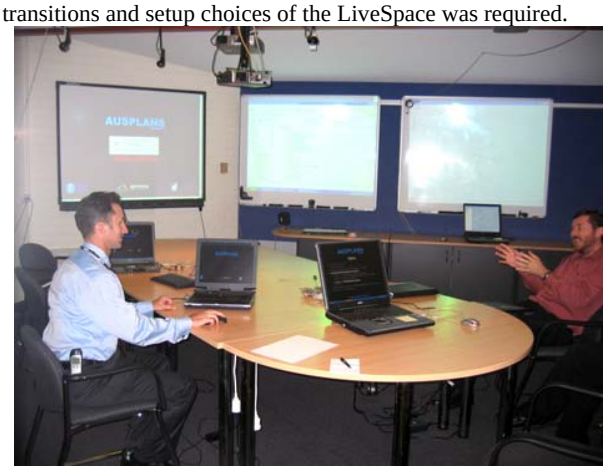
Configuration Support for Ubiquitous Workspaces

Markus Stumptner Bruce Thomas
Advanced Computing Research Centre, University of South Australia
5095 Mawson Lakes (Adelaide) SA, email: {mst|thomas}@cs.unisa.edu.au

Introduction

Ubiquitous workspaces are future media-rich environments that employ new forms of operating systems and services to coordinate and manage interactions between people, multiple display surfaces, information, personal devices, and workspace applications [VER04]. LiveSpaces is a ubiquitous workspaces approach that is addressing how physical spaces such as meeting rooms can be augmented with a range of display technologies, personal information appliances, speech and natural language interfaces, interaction devices and contextual sensors to provide for future interactive/intelligent workspaces. The Figure below shows the LiveSpaces environment that has been set up to support the Commander's Planning Group in the AuSPlanS (distributed joint headquarters planning) project.

Experience with LiveSpaces operation has shown that a significant issue is the actual management of the workspace setup: making sure that, for example, the same (or synchronized) information is shown on the different screens in the LiveSpace, which includes the laptops. In addition, each particular meeting will involve a specific selection of devices (screens, pointing devices, lights, loudspeakers) to be activated that is geared towards its specific needs. Certain devices may be dependent on other ones, leading to a situation where making the changes required to move to the next meeting becomes a major task in itself. A stage was quickly reached where intelligent support for the transitions and setup choices of the LiveSpace was required.



Basic LiveSpace Building Blocks

Configuration is generally defined as the problem of designing a product using a set of predefined components to solve a particular task while taking into account a set of restrictions on how the components can be combined. In more complex cases, the fixed set of components is replaced by the notion of a set of component types, typically organised in a class hierarchy together with a declarative description of how the component types relate, thereby providing an ontology of the domain. To model a comprehensive application environment, we are dealing with five groups of entities: data, devices, applications, users, and processes (meetings).

A **device** is any digitally controllable hardware installation providing a particular service. A special case are platforms corresponding to a server or desktop/laptop machine running applications. **Applications and services** correspond to programs running in the overall workspace. **Users** (user models) exist essentially to store access rights and preferences. **Meeting (Process)** objects will encapsulate the top level state and functionality.

A Livespace environment maps in a reasonable fashion to the configuration ontologies that have been used in the past to describe software or combined software/hardware systems [ASM03], and to constraint-based representations [FLE98]. The intent is to solve the configuration problem in typical fashion by defining the knowledge base (the set of components and the constraints between them), the specification for a desired system (the functionality or the key components required for the system) and then automatically generating a configuration (i.e., set of components and connections) that satisfies all constraints.
The livespace configurator is expected to solve multiple tasks of variable complexity: setup for prespecified meetings, switching between different meetings, and save and restart for a meeting that covers multiple sessions. The declarative representations also permits automated health monitoring of the LiveSpace ubiquitous interface architecture, seamlessly integrating user modeling, hardware and software configuration, as well as generic setups and adaptations of individual meetings.

REFERENCES

[ASM03] T. Asikainen and Timo Soininen and Tomi Männistö, A Koala-Based Ontology for Configurable Software Product Families. Proceedings IJCAI Workshop on Configuration, pp.76—81, 2003.

[FLE98] G. Fleischanderl, G. Friedrich, A. Haselbock, H. Schreiner, M. Stumptner. Configuring large-scale systems with generative constraint satisfaction. IEEE Intelligent Systems, 13(4), Special Issue on Configuration, Juli/August 1998.

[SFH98] M. Stumptner G. Friedrich A. Haselbock. Generative constraint-based configuration of large technical systems. AI EDAM, Volume 12, Issue 04, September 1998, pp 307-320

[VER04] Vernik M.J., Johnson S., Vernik R.J. (2004) "e-Ghosts: leaving virtual footprints in ubiquitous workspaces", Australasian User Interface Conference, Dunedin NZ.

Using Constraint Optimization to Enhance the Diversity in the Set of Computed Configurations

Diego Magro [1]

1 Extended Abstract

Configuration problems may usually have several solutions (valid configurations) and in many cases we are interested in more than one solution. Indeed, in several situations the user requirements may be satisfied by several configured products and more than one option should be presented to the user. However, providing the user with a list containing all the possibilities is usually not a good idea for various reasons. First of all, the number of suitable configurations may be huge and the user could be disoriented by a list containing too many items. Secondly, in the set of configurations fulfilling the requirements, there may be some subsets whose elements are very similar, i.e. such that in any subset, each configuration is only a slight variant of each other configuration in the same subset and it may be not interesting for a user to see all these variants. Moreover, finding all the solutions to a configuration problem may be computationally too expensive. Therefore, in many application domains, it would be useful to have a configurator system that can provide the user with a restricted and manageable set of diverse configurations, i.e. such that each of them is reasonably different from the others.

The problem of improving the diversity in a set of recommended products has been dealt with in the context of content-based recommender systems [1].

Here we discuss a similar approach in the context of configuration. Different from classical recommender system domains, in configuration ones, the set of products is not explicitly available a priori; instead, the set of suitable configurations is computed on the fly, given a general description of the configurable objects (which *implicitly* represents the set of all the valid configurations) and the user requirements. Since, in general, it is not feasible to first compute all the suitable configurations and then select a subset of them to be presented to the user, the approaches designed for classical content-based recommender systems to improve diversity in their suggestions cannot be directly used in the configuration task.

In the last decades, a considerable effort has been devoted by both academic and industrial research in order to automate the configuration task. After the first rule-based configurators, a widely accepted conceptualization for configuration has been defined [10], and different approaches for the knowledge representation and the inference mechanisms have been adopted either based on logic, e.g. [9, 5, 8]), on constraint satisfaction, e.g. [7, 11, 2, 6]), or on both [4].

Here, we refer to the CSP framework and we assume that a configuration problem is encoded into a CSP instance whose solutions are the valid configurations. The problem of finding diverse (and similar) solutions to CSP instances has also been dealt with in [3].

We assume that the similarity degree between two configurations can be measured as a weighted sum of the similarities between the values of the corresponding variables in the two configurations. Therefore, for each variable, we need a weight, representing its importance w.r.t. the similarity assessment, and a similarity function defined over each pair of valid values for that variable; moreover, the similarity degree of a set of configurations is expressed as the average similarity degree of each pair of configurations in the set. Given that, the problem of finding a configuration with the smallest similarity w.r.t. a set of already computed configurations can be easily formalized as a Costraint Optimization Problem. This allows one to define in the configuration domain an algorithm, similar to the *GreedySelection* algorithm presented in [1], which (differently from the *GreedySelection* algorithm) does not assume a given list of items among which selecting those to recommend to the user and such that it dynamically computes the configurations, instead. Even if this approach does not offer the guarantee of always providing the best set of products, we think that it could be interesting to see if it can provide also in configuration domains the same benefits that it has provided in classical content-based recommender systems [1].

[1] University of Torino, Italy, email: magro@di.unito.it

REFERENCES

[1] K. Bradley and B. Smith, 'Improving recommendation diversity', in *Proc. 12th National Conference in Artificial Intelligence Cognitive Science (AICS-01)*, pp. 75–84, (2001).
[2] E. Gelle and M. Sabin, 'Solving methods for conditional constraint satisfaction', in *Proc. IJCAI-03 Configuration WS*, pp. 7–12, (2003).
[3] E. Hebrard, B. Hnich, B. O'Sullivan, and T. Walsh, 'Finding diverse and similar solutions in constraint programming', in *Proc. of the AAAI 05*, (2005).
[4] U. Junker and D. Mailharro, 'The logic of ilog (j)configurator: Combining constraint programming with a description logic', in *Proc. IJCAI-03 Configuration WS*, pp. 13–20, (2003).
[5] D. Magro and P. Torasso, 'Decomposition strategies for configuration problems', *AIEDAM, Special Issue on Configuration*, **17**(1), 51–73, (2003).
[6] B. O'Sullivan, A. Ferguson, and E. Freuder, 'A decision tree learning and constraint satisfaction hybrid for interactive problem solving', in *Proc. IJCAI-05 Configuration WS*, pp. 1–6, (2005).
[7] D. Sabin and E.C. Freuder, 'Configuration as composite constraint satisfaction', in *Proc. Artificial Intelligence and Manufacturing. Research Planning Workshop*, pp. 153–161, (1996).
[8] C. Sinz, A. Kaiser, and W. küchlin, 'Formal methods for the validation of automotive product configuration data', *AIEDAM, Special Issue on Configuration*, **17**(1), 75–97, (2003).
[9] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen, 'Unified configuration knowledge representation using weight constraint rules', in *Proc. ECAI 2000 Configuration WS*, pp. 79–84, (2000).
[10] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen, 'Towards a general ontology of configuration', *AI EDAM*, **12**(4), 383–397, (1998).
[11] M. Veron and M. Aldanondo, 'Yet another approach to ccsp for configuration problem', in *Proc. ECAI 2000 Configuration WS*, pp. 59–62, (2000).